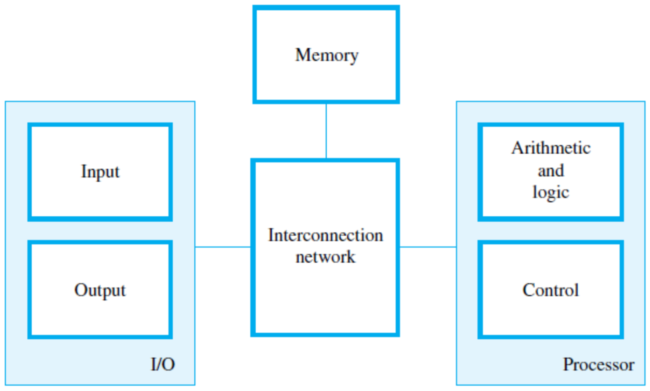
# ECSE 324 REVIEW SUMMARY

**Table of Contents**

# CH 1 Introduction to Computer Organization

## Functional Units of a Computer

- I/O: interacts with the outside world, receives and sends information
- Memory: store both program and data
- Arithmetic and Logic: process information
- Control Unit: coordinate the operations of the computer
- Interconnection Network / Datapath: links devices together and provide path for information exchange



**Embedded System:** For a specific application rather than general computing

Elements:

- Microcontroller: microprocessor and memory
- Converters: ADC and DAC
- Interfaces: sensors and Actuators

Microprocessor vs Microcontroller:

- Microprocessor: only implements a CPU
- Microcontroller: Self-contained system including CPU, memory, peripherals.

Embedded Firmware vs Embedded Software:

- Embedded Firmware: flash memory chip that stores specialized software running in a chip in an embedded device to control its functions
- Embedded Software: any code running on a piece of hardware

Cost of Pure Hardware Products

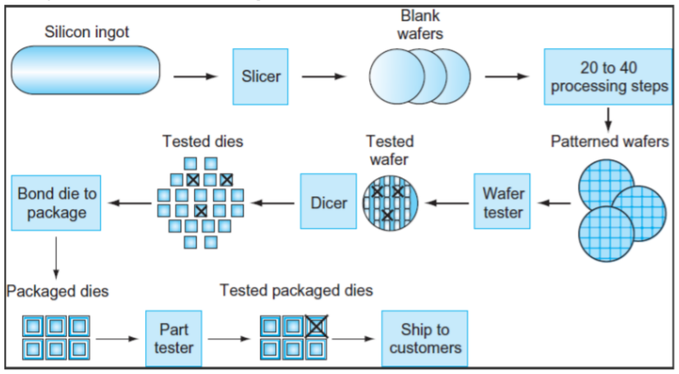
- Cost = NRE + MC * N
  - NRE: Non-recurring engineering or **R&D**
  - MC * N: Manufacturing cost per unit * number of units

Embedded Processor Characteristics:

- Minimum performance
- Strict cost limit
- Strict power consumption limit
- Low tolerance for failure

**Impact of Device Shrinkage**: reduction in transistor size by factor of $x$ results in increase of raw computing power by $x^4$

## Chip Manufacturing



## VLSI Chip Yield

$$Cost\ of\ chip = \frac{Total\ fabrication\ and\ testing\ cost}{\#\ of\ good\ chips}$$

$$Die\ yield = \left(1 + \frac{\frac{defects}{die\ area} * die\ area}{\alpha}\right)^{-\alpha}$$

$$Die\ cost = f(die\ area)^4$$

Clustered VLSI Defects:

- Higher yield

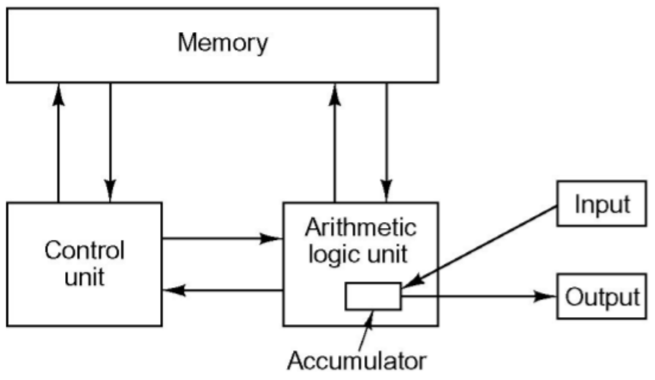
Testing

- Determine and identify faults

- Determine and correct manufacturing process errors causing the fault
- Difficult to generate tests to find all defects

## Computer Architecture

- Von Neumann Machine: stored-program digital computer that uses a processing unit and a single separate storage structure (RAM) to hold both program data and instructions

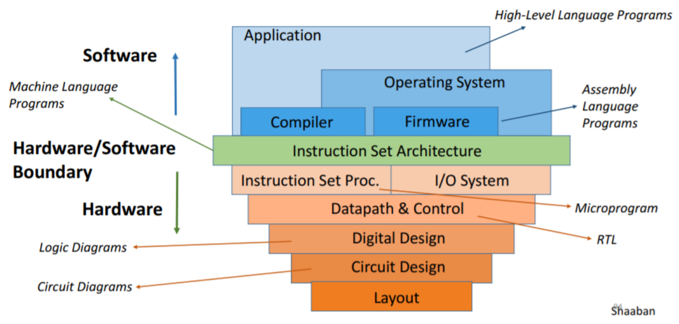


## Engineering Design Approaches

- Refinement: successively refining levels of details (top-down style)
- Modularity: build from smaller pieces and integrate (bottom-up style)

## Hierarchical Design

- Easier transformation from design concept to realization
- Modularity
- Layers of abstraction: easier for task partition and working in parallel



## Microprocessor-based Systems

- CPU: microprocessor, logic circuitry for communication with system bus
- Timing Unit: clock, for proper operation of all hardware
- Memory: store both program data and instructions
- Interrupt Circuitry: mechanism for processor to respond to special external events
- I/O, Peripherals: interface with outside world

## System Software – Compiler

- Translates high-level programs into assembly
- Conversion of source code to object code
  - Source code: code written by programmer using high-level programming languages
  - Object code: code obtained after compiler translation / conversion

## System Software – Operating System

- Infrastructure software component of computer system
- Responsible for management and coordination of activities and sharing of limited resources of computer
  - Handles basic I/O operations
  - Allocates memories to different program
  - Safe sharing of resources between multiple programs
- Acts as host for applications that run on the machine: handles details of operation of hardware for easier programming of software and apps

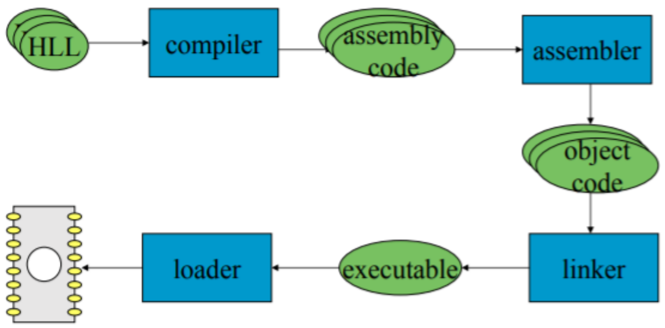
## Assembly Language

- Representation based on mnemonics, implements symbolic representation of numeric machine codes
- Specific to processors and architectures
- Instructions executed sequentially

## High-Level Programming Language

- Strong abstraction from processor details
- Portable and independent of the computer

## Compilation, Assembly, and Linking



## Layered Computer Architecture

- Programs on a higher level either interpreted by interpreter on a lower level machine, or translated to the machine language of a lower machine by a compiler

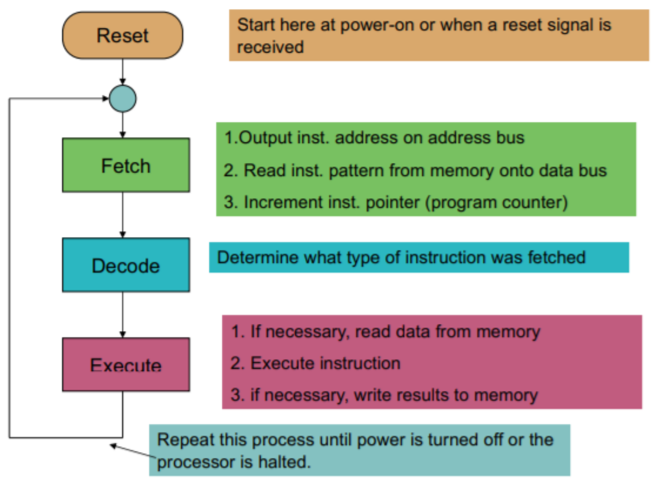
## Program Interpreters

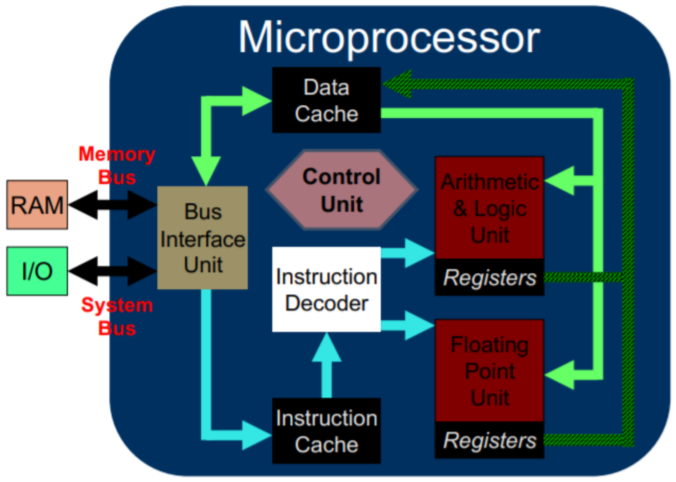
- Imitates CPU function and executes program
- Carries on fetching, examining, and executing program instructions

## Microprocessor Operation

For each instruction:

- Fetch:
  - Fetch next instruction from memory at address pointed to by PC (Contents of PC are transferred to the memory along with a READ control signal)
  - Load the retrieved instruction into IR
  - Increment PC
- Decode:
  - Control unit decodes instruction stored in IR
- Execute:
  - If the instruction requires operands from memory, send the address of the operands to memory with a READ control signal
  - Load the retrieved operands into processor registers
  - ALU performs desired operation on operands and store results in processor registers
  - If results need to be stored in memory, the memory address, operands, and a WRITE control signal is sent to memory.


Reset
Start here at power-on or when a reset signal is received
Fetch
1.Output inst. address on address bus
2. Read inst. pattern from memory onto data bus
3. Increment inst. pointer (program counter)
Decode
Determine what type of instruction was fetched
Execute
1. If necessary, read data from memory
2. Execute instruction
3. if necessary, write results to memory
Repeat this process until power is turned off or the processor is halted.



## Bus Interface Unit

- Receives instruction & data from main memory
- Sends instructions to instruction cache and data to data cache
- Receives the processed data and sends it to the main memory

## Instruction Decoder

- Decodes the received programming instructions into a form understandable by ALU/FPU
- Passes on the decoded instruction to ALU/FPU

## Microarchitecture

- Electrical circuitry of a computer used to implement the instruction set
- Represented by diagrams describing interconnections of various microarchitectural elements of machine: gates, registers, MUXs, LUT, adders, ALU, etc.

- The same instruction set can be implemented on different microarchitectures
- One microarchitecture (block diagram) can have different hardware implementations

**Performance Measurement**
**Execution Time**

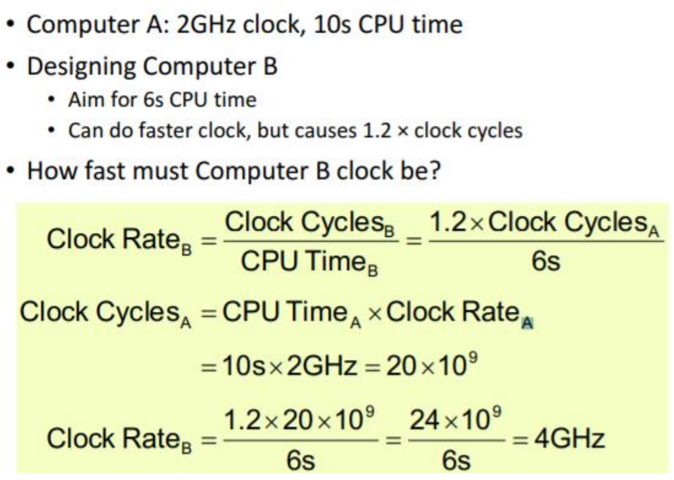
- Elapsed Time: total response time including all aspects (I/O, idle, etc.)
- CPU Time: time spent by CPU on processing a given job

**CPU Clocking**

- Digital hardware operation governed by a constant-rate clock

$$CPU\ time = Clock\ cycles * \frac{Time}{1\ Clock\ cycle} = \frac{Clock\ cycles}{Clock\ rate}$$

<u>Example</u>



- Computer A: 2GHz clock, 10s CPU time
- Designing Computer B
  - Aim for 6s CPU time
  - Can do faster clock, but causes 1.2 × clock cycles
- How fast must Computer B clock be?

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6s}$$

$$\text{Clock Cycles}_A = \text{CPU Time}_A \times \text{Clock Rate}_A$$

$$= 10s \times 2GHz = 20 \times 10^9$$

$$\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6s} = \frac{24 \times 10^9}{6s} = 4GHz$$

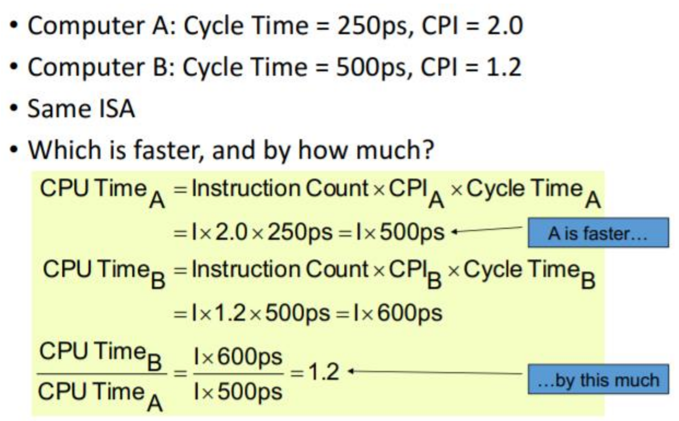
**Instruction Count & CPI**

$$Clock\ cycles = \#\ instructions * \frac{\#\ Cycles}{1\ Instruction}$$

$$CPU\ time = \#\ instructions * CPI * \frac{Time}{1\ Clock\ cycle}$$

$$= \frac{\#\ instructions * CPI}{Clock\ rate}$$

<u>Example</u>



- Computer A: Cycle Time = 250ps, CPI = 2.0
- Computer B: Cycle Time = 500ps, CPI = 1.2
- Same ISA
- Which is faster, and by how much?

$$\text{CPU Time}_A = \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A$$

$$= I \times 2.0 \times 250ps = I \times 500ps$$

A is faster…

$$\text{CPU Time}_B = \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B$$

$$= I \times 1.2 \times 500ps = I \times 600ps$$

$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A} = \frac{I \times 600ps}{I \times 500ps} = 1.2$$

…by this much

**Power CMOS IC**

$$Power = C * V^2 * F$$

**Response Time and Throughput**

- Response time: how long it takes to do a task
- Throughput: total work done per unit time

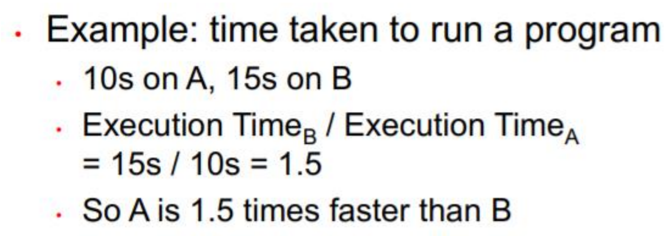
**Relative Performance**

$$Performance = \frac{1}{Execution\ time}$$

- "X is $n$ times faster than Y":

$$\frac{Performance_X}{Performance_Y} = \frac{ExecutionTime_Y}{ExecutionTime_X} = n$$

<u>Example</u>



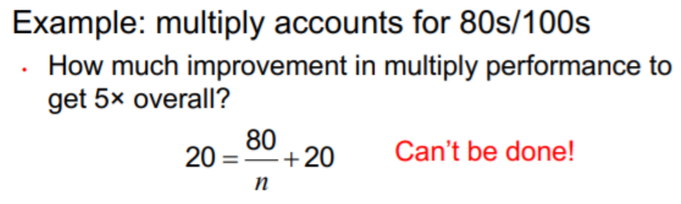
- Example: time taken to run a program
  - 10s on A, 15s on B
  - Execution Time$_B$ / Execution Time$_A$ = 15s / 10s = 1.5
  - So A is 1.5 times faster than B

**Amdahl's Law**

- Component improvement and overall performance improvement

$$T_{improved} = \frac{T_{affected}}{improvement\ factor} + T_{unaffected}$$

<u>Example</u>



Example: multiply accounts for 80s/100s

- How much improvement in multiply performance to get 5× overall?

$$20 = \frac{80}{n} + 20$$

Can't be done!

**Benchmarks**

- Standardized measurement of performance
- MIPS: Millions of instructions per second

$$MIPS = \frac{\#\ instructions}{Execution\ Time * 10^6}$$
$$= \frac{\#\ instructions}{\frac{\#\ instructions * CPI}{Clock\ rate} * 10^6} = \frac{Clock\ rate}{CPI * 10^6}$$

## CH 2 Arithmetic

### Number Representation

With $n$ bits:

- Unsigned Integers:
  - Range: $[0, 2^n - 1]$
- Sign and Magnitude:
  - Range: $[-(2^{n-1} - 1), 2^{n-1} - 1]$
  - MSB used to represent sign
  - Zero is represented twice
- 1's Complement:
  - Range: $[-(2^{n-1} - 1), 2^{n-1} - 1]$
  - Negative numbers obtained by complementing each bit of the corresponding positive number
  - Zero is represented twice
- 2's Complement:
  - Range: $[-2^{n-1}, 2^{n-1} - 1]$
  - Negative numbers obtained by adding "1" to the 1's complement representation of the corresponding positive number

### Addition

- Sum will be correct if result is within range
- Ignore carry out bit for 2's complement for correct result

### Subtraction

- Represent both in 2's complement
- Take again 2's complement of the subtrahend to convert to normal addition

### Overflow

- Result is outside range

Detection of Overflow:

- Unsigned addition: indicated by carry-out bit
- Signed addition:
  - addition of two numbers of opposite polarity will not generate overflow
  - addition of two numbers with same sign and generating result of opposite sign indicates overflow

### Sign Extension

- Positive numbers: append "0" in front
- Negative numbers: append "1" in front

## CH 3 ISA / ARM

### Instruction Set

- Predefined set of instructions which hardware can directly execute.
- Software is composed of the subset of instructions from the instruction set.

### ISA

- Interface between hardware and lowest level software.
- Almost same IS used by different processors regardless of their very different internal designs

### IS Designs

CISC:

- Instructions better optimized
- Memory/cache efficiency improved
- Simplified programming

RISC:

- Architecture with smaller set of instructions
- Each instruction occupy 1 word (32 bits)
- Load/Store Architecture:
  - Operands for ALU must be in registers or given explicitly in instruction
  - Only LDR and STR instructions used to access memory
- Instructions/data stored in memory
- Process register contents initially invalid
- Data transfers are required before arithmetic

### Register Transfer Language

- Describes decoding and sequencing of each ISA instruction for the given physical microarchitecture

### Register Transfer Notation

- […]: contents of location

- ← : transfer (copy) to a destination
- $Mem[A]$: a memory entry at address A
- Right-hand expression always denote a value, left-hand side always names a location

<u>Example</u>:

- $R2 \leftarrow [LOC]$: transfer from LOC in memory to register R2
- $R4 \leftarrow [R2] + [R3]$: add the contents of registers R2 and R3 and place the sum in register R4
- $C \leftarrow [A] + [B]$:
  - Fetch contents of locations A and B
  - Compute sum, transfer results to location C

### PC and IR Registers

- PC: Points to the address of the next instruction in memory
- IR: Holds the machine language version of instruction from memory currently executed

### Details of Instruction Execution

- Two Phase Procedure: fetch and execute
  - Step 1: Fetch next instruction from memory at address pointed to by the PC and write it to the IR
  - Step 2: Decode instruction in the IR (performed in control unit)
  - Step 3: Read operands (from registers for all instructions except Load, or from memory for Load)
  - Step 4: Perform the operation in the ALU (e.g., add two numbers)
  - Step 5: Write Result (to registers for all instructions except Store, memory for Store)
  - Step 6: Update PC to the address of the next instruction (e.g. PC ⟵ [PC] + 4)

### Branching

<u>Example</u>: Adding numbers in a loop

Steps:

- Enter loop
- Perform operation
- Update counter
- Check condition
- Conditional branching

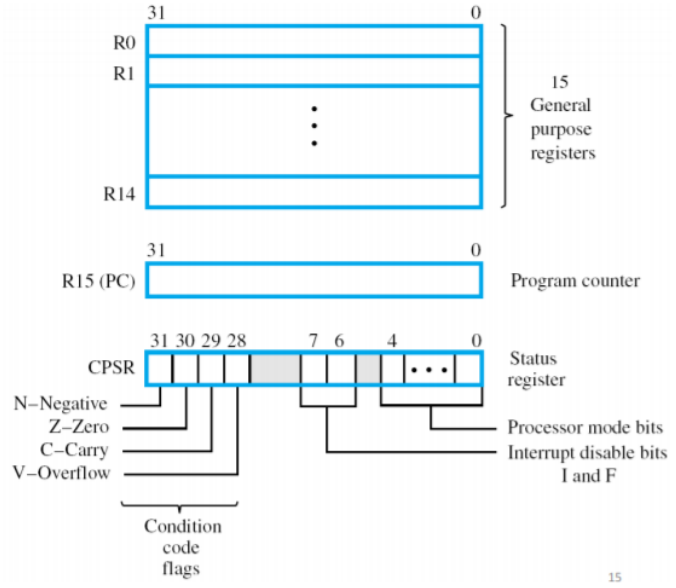
## <u>ARM Processor</u>

### Memory Organization

- Byte-addressable with 32-bit address space
- Little/Big endian addressable
- 1 word = 4 bytes = 32 bits
- Word, half-word, and byte data transfers. Word and half-word transfers must be aligned

### CPU Registers

- 16 32-bit processor registers (R0 – R15)
  - R15: PC
  - R14: LR
  - R13: SP
  - R0-R12: general purpose registers
  - Current program status register (CPSR): holds conditional code flags (N,Z,C,V), 2 interrupt-disable bits, 5 processor mode bits



### Instructions

Arithmetic:

- *OP Rd, Rn, Rm or #offset*
- Immediate values (#offset) in <u>arithmetic</u> are unsigned values in the range 0 to 255 (8 bits)

| | | |
|---|---|---|
| ADC | add two 32-bit values and carry | $Rd = Rn + N +$ `carry` |
| ADD | add two 32-bit values | $Rd = Rn + N$ |
| RSB | reverse subtract of two 32-bit values | $Rd = N - Rn$ |
| RSC | reverse subtract with carry of two 32-bit values | $Rd = N - Rn -$ `!(carry flag)` |
| SBC | subtract with carry of two 32-bit values | $Rd = Rn - N -$ `!(carry flag)` |
| SUB | subtract two 32-bit values | $Rd = Rn - N$ |

<u>Example</u>: Addition

- Assembly: *ADD R0, R2, R4*
- Operation: $R0 \leftarrow [R2] + [R4]$

<u>Example</u>: Subtraction

- Assembly: *SUB R0, R3, #17*
- Operation: $R0 \leftarrow [R3] - 17$

<u>Example</u>: Multiplication

- Assembly: *MUL R0, R1, R2*

- Operation: $R0 \leftarrow [R1] \times [R2]$

Example: Add and Multiply

- Assembly: *MLA R0, R4, R5, R6*
- Operation: $R0 \leftarrow ([R4] \times [R5]) + [R6]$

Logic:

- Instruction: AND Rd, Rn, Rm
- Operation: Bitwise logical AND of Rn and Rm and store results in Rd
- Instruction: ORR
- Operation: bit-wise logical OR
- Instruction: EOR
- Operation: bit-wise logical XOR
- Instruction: BIC Rd, Rn, Rm
- Operation: Bits of Rm complemented then ANDed with Rn and stored in Rd

```
•AND   R0, R1, R2    @ R0 = R1 and R2
•ORR   R0, R1, R2    @ R0 = R1 or  R2
•EOR   R0, R1, R2    @ R0 = R1 xor R2
•BIC   R0, R1, R2    @ R0 = R1 and (~R2)
```

TST Test Instruction

- Instruction: TST Rn, Rm or #value
- Operation: bit-wise logical AND of two operands, sets the conditional code flags. Z flag is updated by <u>result</u> of the AND operation.

Example:

- Instruction: TST R0, #1
  - Z = 1 if LSB of R0 is 0
  - Z = 0 if LSB of R0 is 1

TEQ Test Instruction

- Instruction: TEQ Rn, Rm or #value
- Operation: bit-wise logical XOR of the two operands, conditional flag Z set by the result of the XOR operation

Example:

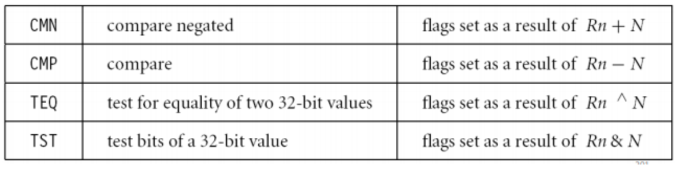
- Instruction: TEQ R0, #1
  - Z = 1 if R0 is 1
  - Z = 0 if R0 is not 1

CMP Instruction

- Instruction: CMP Rn, Rm or #value
- Operation: $[Rn] - [Rm]$ and updates conditional code flags

CMN Instruction

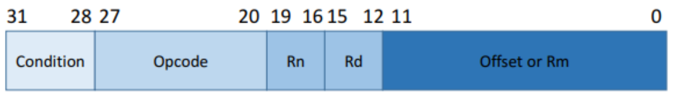
- Instruction: CMN Rn, Rm or #value
- Operation: $[Rm] - [Rn]$ and updates conditional code flags



| | | |
|---|---|---|
| CMN | compare negated | flags set as a result of $Rn + N$ |
| CMP | compare | flags set as a result of $Rn - N$ |
| TEQ | test for equality of two 32-bit values | flags set as a result of $Rn$ ^ $N$ |
| TST | test bits of a 32-bit value | flags set as a result of $Rn$ & $N$ |

**Machine Code**

- Instructions in binary string divided into fields, each encoding part of assembly instruction



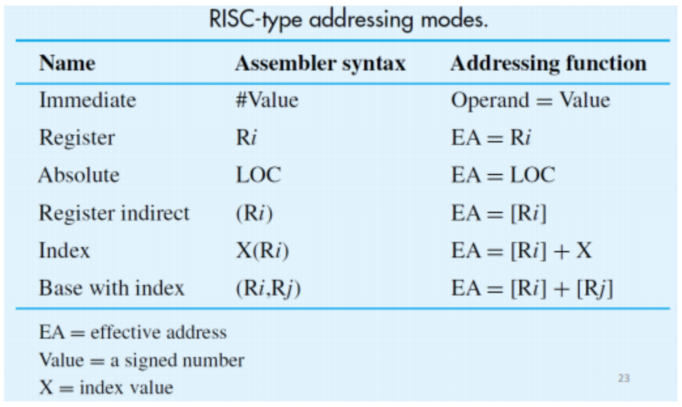
- Register fields *Rn* and *Rd* are 4-bit wide since there are 16 general purpose registers in total

Example:

- Assembly: *ADD R4, R5, #24*
- 32-bit machine instruction: XXXX 00101000 0110 0101 000000011000

**Addressing Modes**

- Provide compiler with different modes of specifying operand location
- The effective address (EA) of a memory operand is the <u>sum</u> of the <u>contents</u> of a base register and a signed offset
- Offset in <u>addressing mode</u> is either:
  - Immediate: 12-bit immediate value added/subtracted from base *Rn*
  - Index: contents of second register *Rm*, can be scaled before use



RISC-type addressing modes.

| Name | Assembler syntax | Addressing function |
|---|---|---|
| Immediate | #Value | Operand = Value |
| Register | R*i* | EA = R*i* |
| Absolute | LOC | EA = LOC |
| Register indirect | (R*i*) | EA = [R*i*] |
| Index | X(R*i*) | EA = [R*i*] + X |
| Base with index | (R*i*,R*j*) | EA = [R*i*] + [R*j*] |

EA = effective address
Value = a signed number
X = index value

23

Register Mode

- Instruction: *ADD R1, R2, R3*
- All operands in registers

Immediate Mode

- Instruction: *ADD R1, R2, #15*
- Immediate value 15 explicitly defined in instruction

**Indexed Addressing Mode – Shifting**

- The second source operand can be shifted or rotated before use
- Shifting by $k$ bits to the left = multiplication by $2^k$
- Shifting by $k$ bits to the right = division by $2^k$
- LSL/LSR fill new positions with zeros
- ASR shifts right while preserving the sign bit (MSB)

Indexed Mode - Shifting

- Instruction: *ADD R0, R1, R5, LSL #4*
- Operation: $R0 \leftarrow [R1] + 16 \times [R5]$

**Load/Store Addressing Mode**

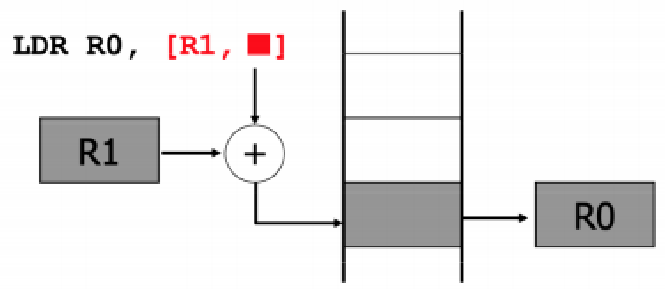
- Offset Mode: EA = [base register Rn] + offset
- Pre-Indexed Mode: EA = [base register Rn] + offset but base register *Rn* is updated with EA after operation
- Post-Indexed Mode: EA = [base register *Rn*] and *Rn* is updated with base + offset

**Pre-Index**

Pre-Indexed Addressing Mode

- Instruction: *LDR Rd, [Rn, #offset]*
- Operation: $Rd \leftarrow [[Rn] + offset]$
- Instruction: *LDR Rd, [Rn, Rm]*
- Operation: $Rd \leftarrow [[Rn] + [Rm]]$

```
LDR  R0, [R1, #4]    @ R0=mem[R1+4]
                     @ R1 unchanged
```



Pre-Indexed with Writeback

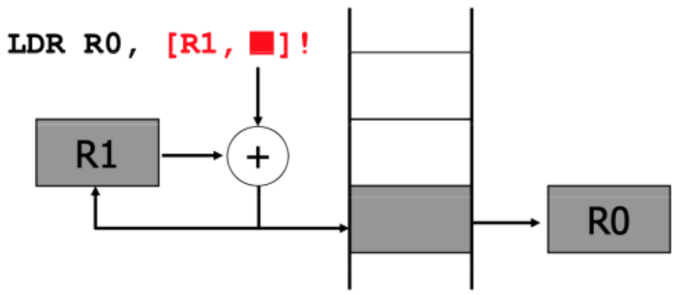
- Instruction: *LDR Rd, [Rn, #offset]!*
- Operation:
  - $Rd \leftarrow [[Rn] + offset]$
  - $Rn \leftarrow [Rn] + offset$

Example:

- Instruction: *LDR R4, [R2, -R3, LSL #4]!*
- Operation:
  - $R4 \leftarrow [[R2] - [R3] \times 2^4]$
  - $R2 \leftarrow [R2] - [R3] \times 2^4$

```
LDR  R0, [R1, #4]!   @ R0=mem[R1+4]
                     @ R1=R1+4
```

No extra time; Fast;



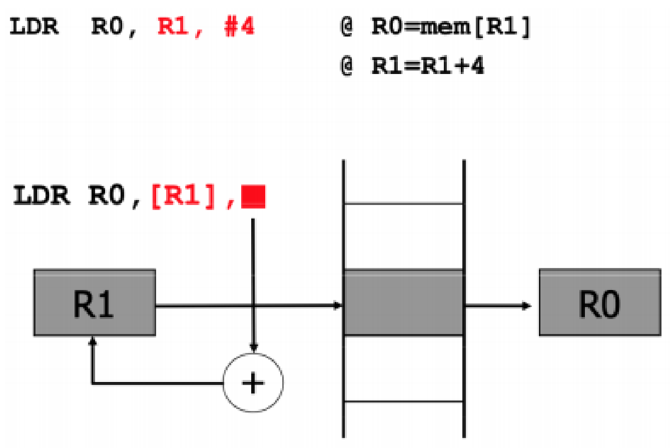
Relative Addressing Mode

- Instruction: *LDR Rd, LOC*
- Operation: $Rd \leftarrow [LOC] = [[PC] + offset]$
- The offset is calculated by the assembler

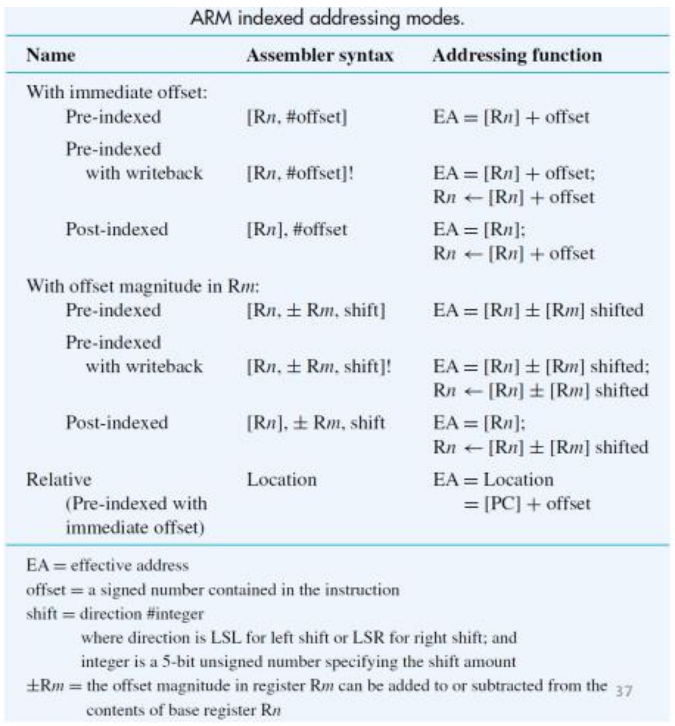
**Post-Index**

Post-Indexed Addressing Mode

- Instruction: *LDR Rd, [Rn], #offset*
- Operation:
  - $Rd \leftarrow [[Rn]]$
  - $Rn \leftarrow [Rn] + offset$
- Instruction: *LDR Rd, [Rn], Rm*
- Operation:
  - $Rd \leftarrow [[Rn]]$
  - $Rn \leftarrow [Rn] + [Rm]$



```
LDR   R0, R1, #4      @ R0=mem[R1]
                      @ R1=R1+4
```



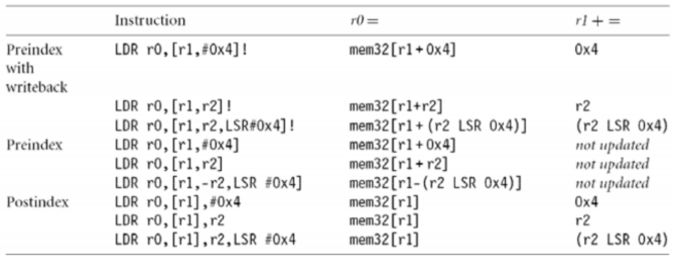
ARM indexed addressing modes.

| Name | Assembler syntax | Addressing function |
|---|---|---|
| With immediate offset: | | |
| Pre-indexed | [R*n*, #offset] | EA = [R*n*] + offset |
| Pre-indexed with writeback | [R*n*, #offset]! | EA = [R*n*] + offset; R*n* ← [R*n*] + offset |
| Post-indexed | [R*n*], #offset | EA = [R*n*]; R*n* ← [R*n*] + offset |
| With offset magnitude in R*m*: | | |
| Pre-indexed | [R*n*, ± R*m*, shift] | EA = [R*n*] ± [R*m*] shifted |
| Pre-indexed with writeback | [R*n*, ± R*m*, shift]! | EA = [R*n*] ± [R*m*] shifted; R*n* ← [R*n*] ± [R*m*] shifted |
| Post-indexed | [R*n*], ± R*m*, shift | EA = [R*n*]; R*n* ← [R*n*] ± [R*m*] shifted |
| Relative (Pre-indexed with immediate offset) | Location | EA = Location = [PC] + offset |

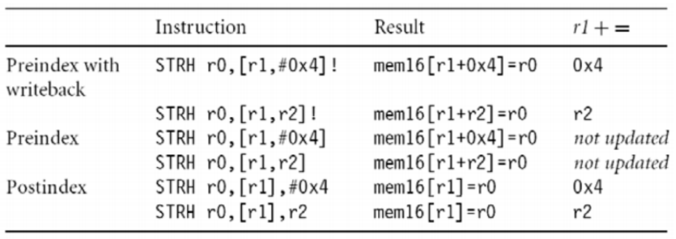
EA = effective address
offset = a signed number contained in the instruction
shift = direction #integer
where direction is LSL for left shift or LSR for right shift; and
integer is a 5-bit unsigned number specifying the shift amount
±R*m* = the offset magnitude in register R*m* can be added to or subtracted from the contents of base register R*n*

## Comparison of Addressing Modes

| Instruction | R0= | R1= |
|---|---|---|
| LDR R0, [R1, R2] | Mem[R1 + R2] | R1 |
| LDR R0, [R1, R2]! | Mem[R1 + R2] | R1 + R2 |
| LDR R0, [R1], R2 | Mem[R1] | R1 + R2 |



| | Instruction | *r0* = | *r1* + = |
|---|---|---|---|
| Preindex with writeback | LDR r0,[r1,#0x4]! | mem32[r1+0x4] | 0x4 |
| | LDR r0,[r1,r2]! | mem32[r1+r2] | r2 |
| | LDR r0,[r1,r2,LSR#0x4]! | mem32[r1+(r2 LSR 0x4)] | (r2 LSR 0x4) |
| Preindex | LDR r0,[r1,#0x4] | mem32[r1+0x4] | *not updated* |
| | LDR r0,[r1,r2] | mem32[r1+r2] | *not updated* |
| | LDR r0,[r1,-r2,LSR #0x4] | mem32[r1-(r2 LSR 0x4)] | *not updated* |
| Postindex | LDR r0,[r1],#0x4 | mem32[r1] | 0x4 |
| | LDR r0,[r1],r2 | mem32[r1] | r2 |
| | LDR r0,[r1],r2,LSR #0x4 | mem32[r1] | (r2 LSR 0x4) |



| | Instruction | Result | *r1* + = |
|---|---|---|---|
| Preindex with writeback | STRH r0,[r1,#0x4]! | mem16[r1+0x4]=r0 | 0x4 |
| | STRH r0,[r1,r2]! | mem16[r1+r2]=r0 | r2 |
| Preindex | STRH r0,[r1,#0x4] | mem16[r1+0x4]=r0 | *not updated* |
| | STRH r0,[r1,r2] | mem16[r1+r2]=r0 | *not updated* |
| Postindex | STRH r0,[r1],#0x4 | mem16[r1]=r0 | 0x4 |
| | STRH r0,[r1],r2 | mem16[r1]=r0 | r2 |

## Load/Store Byte and Halfword

- LDRB: load byte, zero padded to 32-bits
- LDRH: load halfword, zero padded to 32-bits
- LDRSB: load byte with sign extension to 32-bits
- LDRSH: load halfword with sign extension to 32-bits
- STRB: store lower-order byte of register
- STRH: store lower-order halfword of register

Note: Data transfers must be aligned

## Multiple Register Load/Store

- ARM processor requires $2 + Nt$ cycles
  - N: number of words
  - t: time for a word for sequential access
- Instruction: *LDMIA R1, {R0, R2, R5}*
- Operation:

| RTN | Unknown Notation |
|---|---|
| $R0 \leftarrow [[R1]]$ | R0 = mem[R1] |
| $R2 \leftarrow [[R1] + 4]$ | R2 = mem[R1 + 4] |
| $R5 \leftarrow [[R1] + 8]$ | R5 = mem[R1 + 8] |

- Assembler arranges list of registers in increasing index order. Lowest index corresponds to lowest memory address
- Suffix for memory direction:
  - IA: increment after data transfer
  - IB: increment before data transfer
  - DA: decrement after data transfer
  - DB: decrement before data transfer

## Equivalent Addressing Mode Suffixes

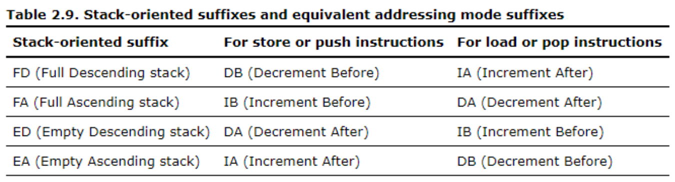


**Table 2.9. Stack-oriented suffixes and equivalent addressing mode suffixes**

| Stack-oriented suffix | For store or push instructions | For load or pop instructions |
|---|---|---|
| FD (Full Descending stack) | DB (Decrement Before) | IA (Increment After) |
| FA (Full Ascending stack) | IB (Increment Before) | DA (Decrement After) |
| ED (Empty Descending stack) | DA (Decrement After) | IB (Increment Before) |
| EA (Empty Ascending stack) | IA (Increment After) | DB (Decrement Before) |

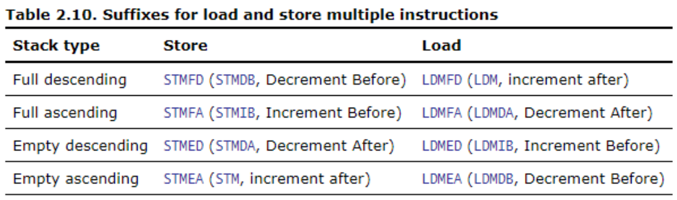
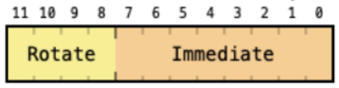


**Table 2.10. Suffixes for load and store multiple instructions**

| Stack type | Store | Load |
|---|---|---|
| Full descending | STMFD (STMDB, Decrement Before) | LDMFD (LDM, increment after) |
| Full ascending | STMFA (STMIB, Increment Before) | LDMFA (LDMDA, Decrement After) |
| Empty descending | STMED (STMDA, Decrement After) | LDMED (LDMIB, Increment Before) |
| Empty ascending | STMEA (STM, increment after) | LDMEA (LDMDB, Decrement Before) |

## Constants in Assembly Instructions

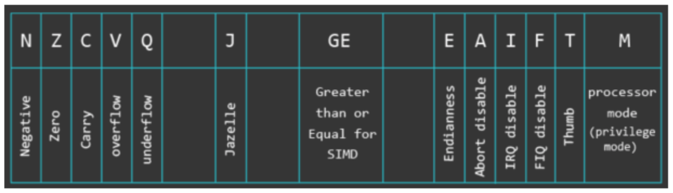


- 12-bit immediate field includes 8-bit number and 4-bit rotation to the right (16 right rotations of 8-bit) number to form 32-bit number

## Move Instruction

- Instruction: *MOV Rd, Rm*
- Operation: $Rd \leftarrow [Rm]$
- Instruction: *MOV Rd, #value*
- Operation: $Rd \leftarrow value$
- Instruction: *MVN Rd, Rm or #value*
- Operation: loads the bit complement of Rm or value into Rd

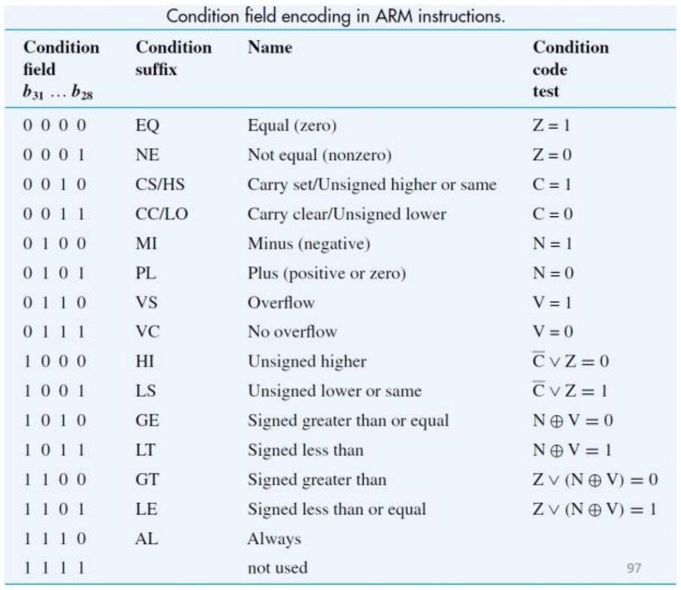
## CPSR



- N bit is 1 if result is negative
- Z bit is 1 if result is 0

## Branch Instruction

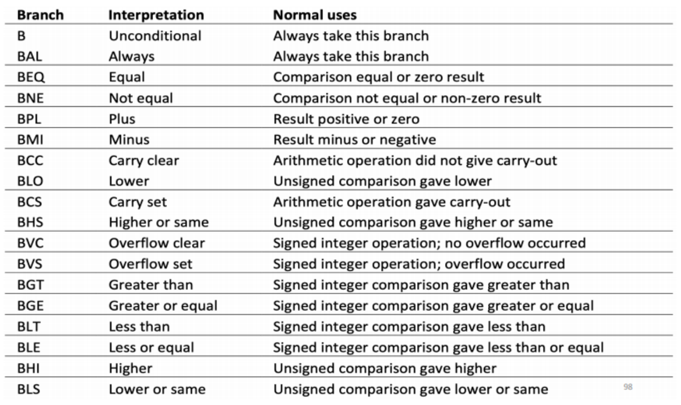
- Instruction: B{condition} LOCATION
- Operation: branch to instruction stored in memory location LOCATION if CPSR satisfies the condition; else do not branch and execute *fallthrough* instruction (instruction immediately after the branch instruction)
- Branch instruction contains 24-bit 2's complement offset, which is left-shifted by 2 bits first then sign extended to 32-bits, and added to PC
- The offset is actually 26 bits but represented with 24 bits because all instructions are word aligned and occupy 4 bytes. With byte-addressable memory, memory locations for instructions change 4 bytes at a time, the last two bits will always be 0 and can be ignored.
- Offset calculated by assembler



Condition field encoding in ARM instructions.

| Condition field $b_{31} \ldots b_{28}$ | Condition suffix | Name | Condition code test |
|---|---|---|---|
| 0 0 0 0 | EQ | Equal (zero) | $Z = 1$ |
| 0 0 0 1 | NE | Not equal (nonzero) | $Z = 0$ |
| 0 0 1 0 | CS/HS | Carry set/Unsigned higher or same | $C = 1$ |
| 0 0 1 1 | CC/LO | Carry clear/Unsigned lower | $C = 0$ |
| 0 1 0 0 | MI | Minus (negative) | $N = 1$ |
| 0 1 0 1 | PL | Plus (positive or zero) | $N = 0$ |
| 0 1 1 0 | VS | Overflow | $V = 1$ |
| 0 1 1 1 | VC | No overflow | $V = 0$ |
| 1 0 0 0 | HI | Unsigned higher | $\overline{C} \vee Z = 0$ |
| 1 0 0 1 | LS | Unsigned lower or same | $\overline{C} \vee Z = 1$ |
| 1 0 1 0 | GE | Signed greater than or equal | $N \oplus V = 0$ |
| 1 0 1 1 | LT | Signed less than | $N \oplus V = 1$ |
| 1 1 0 0 | GT | Signed greater than | $Z \vee (N \oplus V) = 0$ |
| 1 1 0 1 | LE | Signed less than or equal | $Z \vee (N \oplus V) = 1$ |
| 1 1 1 0 | AL | Always | |
| 1 1 1 1 | | not used | |

97



| Branch | Interpretation | Normal uses |
|---|---|---|
| B | Unconditional | Always take this branch |
| BAL | Always | Always take this branch |
| BEQ | Equal | Comparison equal or zero result |
| BNE | Not equal | Comparison not equal or non-zero result |
| BPL | Plus | Result positive or zero |
| BMI | Minus | Result minus or negative |
| BCC | Carry clear | Arithmetic operation did not give carry-out |
| BLO | Lower | Unsigned comparison gave lower |
| BCS | Carry set | Arithmetic operation gave carry-out |
| BHS | Higher or same | Unsigned comparison gave higher or same |
| BVC | Overflow clear | Signed integer operation; no overflow occurred |
| BVS | Overflow set | Signed integer operation; overflow occurred |
| BGT | Greater than | Signed integer comparison gave greater than |
| BGE | Greater or equal | Signed integer comparison gave greater or equal |
| BLT | Less than | Signed integer comparison gave less than |
| BLE | Less or equal | Signed integer comparison gave less than or equal |
| BHI | Higher | Unsigned comparison gave higher |
| BLS | Lower or same | Unsigned comparison gave lower or same |

98

## Assembler Directives

- Instructions for assembler on how to assemble
  - *.text*: marks beginning of code
  - *.end*: marks end of code
  - *.global* ___: marks visible outside object file
  - *.word*: reserve space for words in memory

## LDR with Constant

- Instruction: LDR Rd, =value
- Operation: assembler places the constant "value" in memory after where the instructions are stored, and computes the offset from PC. During execution, the value is retrieved by

  LDR Rd, [PC, #offset]

  Where value = [[PC] + offset]

Example

```
        .text
        .global     _start
-start: LDR  R1, =0xF0F0F0F0
        .end
```

| address | [address] | code |
|---|---|---|
| 0x00000000 | 0xE51F1004 | LDR R1, [PC, #-4] |
| 0x00000004 | 0xF0F0F0F0 | .word 0xF0F0F0F0 |

**Subroutine**

- A block of instructions

Subroutine Linkage

- Instruction: BL SUBADDRESS
- Operation: the address of the next instruction (return address) is stored in the link register LR (R14)
- Instruction: BX LR
- Operation: subroutine branches back to the address stored in LR

**Stack**

- LIFO: last-in-first-out
- Descending stack: top of stack grows toward lower memory addresses
- Ascending stack: top of stack grows toward higher memory addresses
- Instructions:

$$PUSH\ \{reg\ list\} = STMDB\ sp!,\ \{reg\ list\}$$
$$POP\ \{reg\ list\} = LDMIA\ sp!,\ \{reg\ list\}$$

**Stack Pointer**

- SP (R13) points to top of processor stack
- Instruction: PUSH {Rj}
- Operation:

| ASM | RTN |
|---|---|
| SUB SP, SP, #4 | $SP \leftarrow [SP] - 4$ |
| STR Rj, [SP] | $[SP] \leftarrow [Rj]$ |

- Instruction: POP {Rj}
- Operation:

| ASM | RTN |
|---|---|
| LDR Rj, [SP] | $Rj \leftarrow [[SP]]$ |
| ADD SP, SP, #4 | $SP \leftarrow [SP] + 4$ |

**Subroutine Nesting**

- Calling a subroutine within a subroutine
- LR should be pushed onto stack first to avoid being overwritten

**Reset and Initialization**

- Reset: immediately after power is applied and when "reset" signal is asserted
- SP initialized to 32-bit value at location 0 within ROM
- PC initialized to 32-bit value at location 4 within ROM
- LR initialized to 0xFFFFFFFF

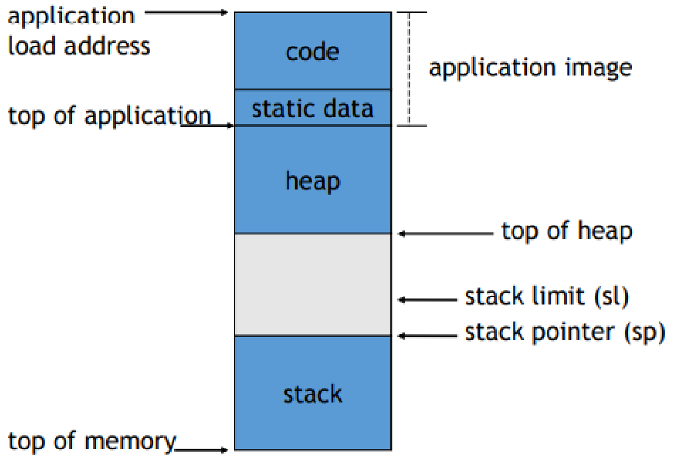
**Parameter Passing**

- R0-R3 used to pass first 4 input parameters
- Additional parameters passed via stack
- Return parameter placed in R0
- Functions free to modify R0-R3
- R4-R11 values must be preserved when returning

**Stack Frame**

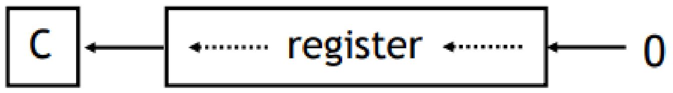
- Location at top of processor stack used as subroutine private workspace
- Frame Pointer (FP)

**Standard ARM C Program Address Space**

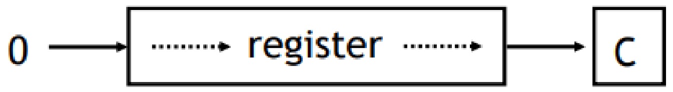


**Shift & Rotate Instructions**

Logic Shift Left



Logic Shift Right



Rotate Right

## CH 4 Memory Organization

### Memory

- Large storage hardware element for keeping data and instructions used by CPU for program execution
- Non-volatile memory: preserves contents when power off
- Memory Access Time: time from initiation to completion of a transfer
- Memory Cycle Time: minimum time delay between successive transfers
- Random-Access Memory: access time is same and independent of location

EPROM

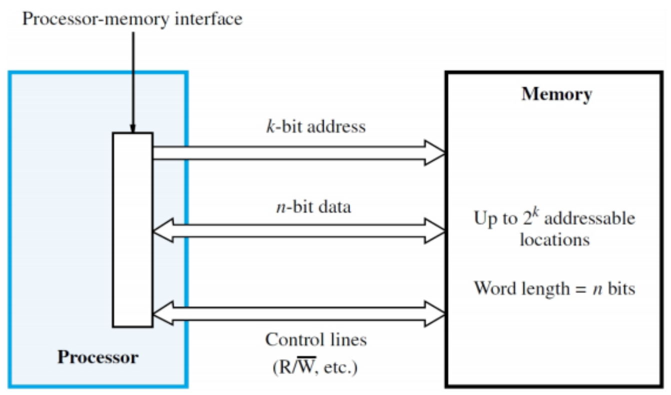
- Erasable programmable read-only memory
- Erasable by UV light

EEPROM

- Electrically erasable programmable read-only memory
- Need 24V to erase

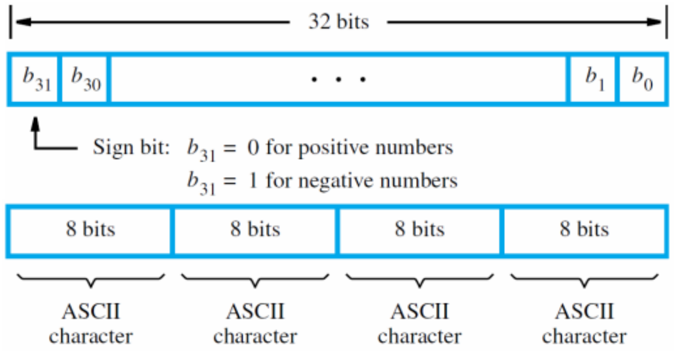
### Memory Operations

- Memory contains data and program instructions
- Control circuits initiate transfer of data and instructions between processor and memory
- READ: memory retrieves contents at address location given by processor
- WRITE: memory overwrites contents at given location with given data


Processor-memory interface
k-bit address
n-bit data
Control lines (R/W̄, etc.)
Processor
Memory
Up to 2^k addressable locations
Word length = n bits

### Word and Byte Encoding

- Word length: 32 bits, four 8-bit bytes

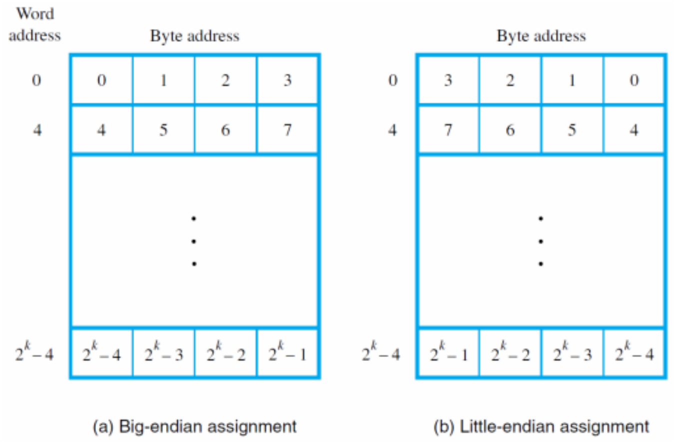


### Memory Addressing

- K-bits in address: $2^k$ memory locations constitute the address space, sets the memory size
- Example:
  - K = 20: $2^{20}$ or 1M locations addressable
  - K = 32: $2^{32}$ or 4G locations addressable
- Byte-addressable

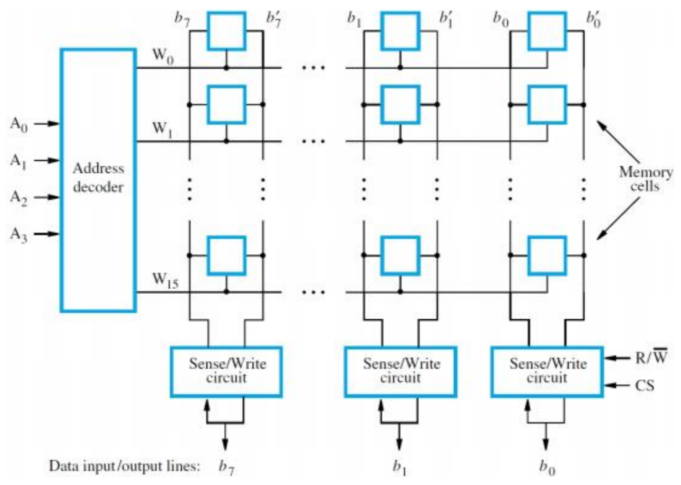
### Endianness

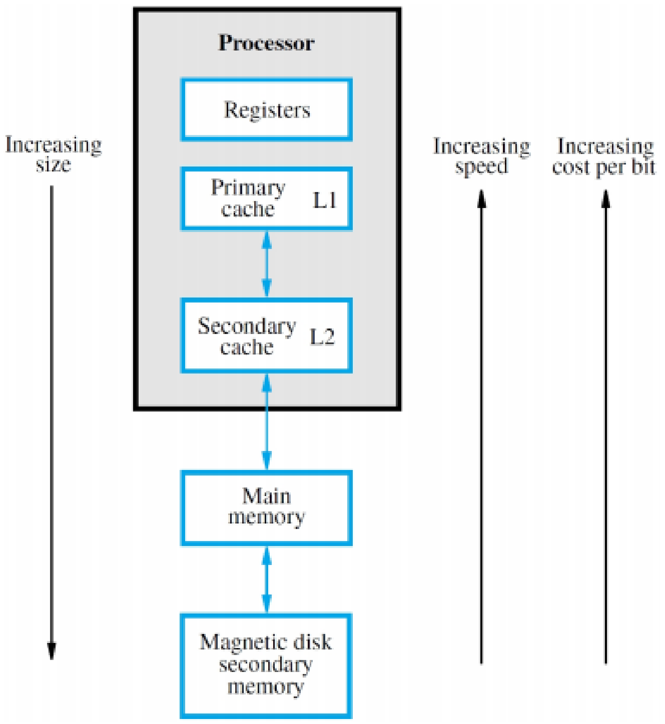
- Two ways to assign byte address across words


Word address
Byte address
0 | 0 1 2 3
4 | 4 5 6 7
2^k−4 | 2^k−4 2^k−3 2^k−2 2^k−1
(a) Big-endian assignment
Byte address
0 | 3 2 1 0
4 | 7 6 5 4
2^k−4 | 2^k−1 2^k−2 2^k−3 2^k−4
(b) Little-endian assignment

### Semiconductor RAM Memories

- Word lines: rows
- Bit lines: columns
- $R/\overline{W}$: specifies read or write operation
- CS: selects the particular chip
- Addressing encoding allows address space $L = \log_2 N$ for N word memory





Static RAM (SRAM)

- 6 transistors per cell
- Faster access time
- No need for periodic refreshing

Dynamic RAM (DRAM)

- 1 transistor and capacitor
- Slower access time
- State is presence/absence of capacitor charge, charge leaks away and must be refreshed

**Boot Loader**

- Simple program initializing operations of processor
- Circuit initializes PC to start address 0
- All processor memory requests from address 0 will be directed to non-volatile memory storing hard-wired program (boot loader)

**Cache and Virtual Memory**

- Main memory: slower than processor
- Cache memory: smaller and faster memory
- Virtual memory: provides larger apparent size by transparently using secondary storage
- Cache handled by hardware
- Virtual memory handled by OS
- Both hidden from programmer

**Caches and Locality of Reference**

- Effectiveness based on locality of reference
- Typical program behaviour involves executing instructions in loops and accessing array data
- Temporal locality: instructions/data that have been recently accessed are likely to be again
- Spatial locality: nearby instructions or data are likely to be accessed after current access
- Transfer block of data with multiple adjacent words from memory to cache
- Mapping function: determines where a block from memory is to be stored in cache
- Replacement algorithm: which block to remove when cache is full

**Cache Operation**

- Processor issues Read/Write directly
- Control circuits first checks cache

**Cache Hit**

- Target memory address found in cache
- Read hit: cache provides desired information
- Write hit: write-through protocol or write-back protocol

**Cache Miss**

- Read miss: block with desired word is transferred from main memory to cache, then the word is sent to the processor
- Load-Through Protocol: the word sent to processor as soon as it is read from main memory
- Write miss

Write-Through Protocol

- Hit: update both cache and memory
- Miss: information written to main memory directly

Write-Back Protocol

- Hit: only update cache and update memory later when block is replaced
- Miss: first transfer block containing the addressed word into cache then overwrite specific word in cache, update memory later when block is replaced
- Better when same location is written repeatedly

**Cache Organization**

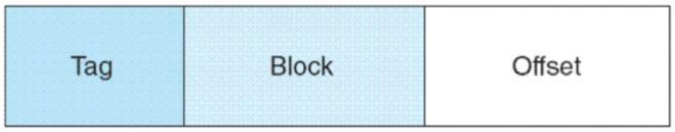
- Valid Cache Bit: indicates whether cache line stores valid data
- Tag Bits: uniquely identifies the block stored in cache line
- Cache Size = Sets * Blocks * Bytes per block

Direct-Mapped Cache

- Mapping function: If cache has N blocks, block X of main memory maps to cache block:

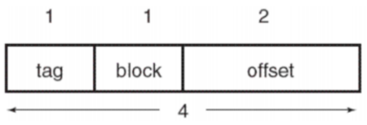
$$Y = X \bmod N$$

- Address divided into 3 fields: tag, block, word
- Retrieving a word stored in cache:
  - Extract "block" field from address
  - Verify contents by checking valid bit and comparing tag field
  - If verified, "word" offset used to access the word in the block
- Each memory block is mapped uniquely to a cache block (cache line)
- No need to search cache
- Cache Miss: fetch block from memory based on the "block" field and load into cache to replace block

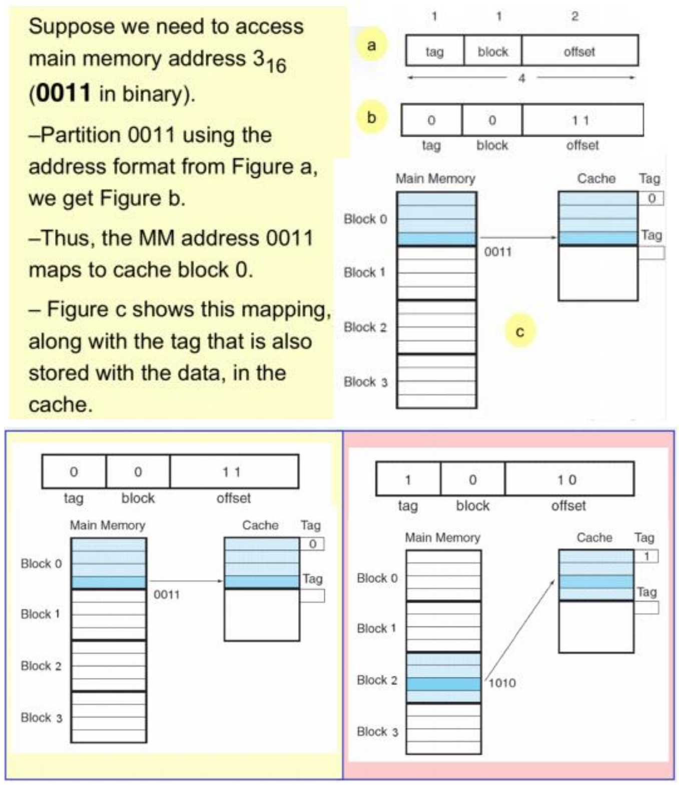


Example: Address fields

- Word-addressable main memory consisting of 4 blocks, and cache with 2 blocks
- Each block is 4 words (word length can be anything)
- For mapping we split main memory address into 3 fields:
  - Each block is 4 words, so the offset field must have 2 bits
  - There are 2 blocks in cache, so the block field must contain 1 bit
  - This leaves 1 bit for the tag



Example:



Suppose we need to access main memory address $3_{16}$ (**0011** in binary).

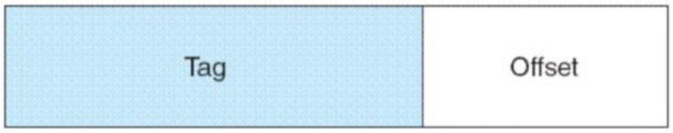
–Partition 0011 using the address format from Figure a, we get Figure b.

–Thus, the MM address 0011 maps to cache block 0.

– Figure c shows this mapping, along with the tag that is also stored with the data, in the cache.

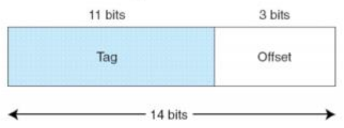
Fully-Associative Mapping

- Block can be anywhere in cache
- No need for "block" field in address
- For hit/miss, the tag field of all blocks within cache are compared simultaneously (in parallel) to retrieve data quickly. Requires costly hardware.



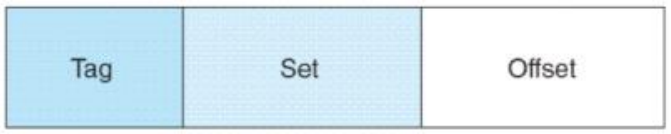
Example:

- We have 14-bit main memory address and a cache with 16 blocks, each block of size 8
- The field format of a memory reference is:



Set-Associative Cache

- Retrieving a word:
  - Extract "set" field from memory address and find set
  - Check valid bit and compare all tags within the set
  - If found, use word offset to find word within block



- If there is only 1 cache line (block) per set: this is equivalent to a direct-mapped cache
- If there is only 1 set in total: this is equivalent to a fully-associative mapped cache
- Lower miss ratio than direct-mapped cache but cheaper implementation than fully-associative cache
- N-way set associative mapping: N cache lines (blocks) per set

Example:

- Assume the following data:
- Main memory size = 2 KB
- Block size B = 8 bytes = $2^3$ bytes
- Cache size C = 64 bytes = $2^6$ bytes
- Set size L = 2
- Number of sets in cache S = 4 = $2^2$
- Verification: C = S x L x B = 4 x 2 x 8 bytes = 64 bytes

Example:

- We are using 2-way set associative mapping with a word-addressable main memory of $2^{14}$ words and a cache with 16 blocks, where each block contains 8 words
  - Cache has a total of 16 blocks, and each set has 2 blocks, then there are 8 sets in cache
  - Thus, the set field is 3 bits, the offset field is 3 bits and the tag field is 8 bits

**Replacement Policy**

- Least-Recently Used (LRU) policy:
  - Remove cache line (block) least recently accessed, utilizes temporal locality
  - Disadvantage: complexity. LRU needs to maintain access history for each block within set.
- FIFO policy: regardless of when the cache line (block) was accessed, remove the one that's in cache for the longest time
- Random policy: randomly replaces block with new block
- Replacement policy requires additional hardware and memory to remember cache access order

**Cache Performance Measurement**

- Effective Access Time (EAT): weighted average that takes into account the hit ratio and relative access time of successive levels of memory
- EAT = Hit ratio * Cache access time + (1 - Hit ratio) * Main memory access time

Example

- A computer system has a main memory access time of 200 ns supported by a cache having a 10 ns access time ad a hit rate of 99%
- Suppose access to cache and main memory occurs concurrently (the access overlap)
- The EAT is: 0.99(10 ns) + 0.01(200 ns) = 9.9 ns + 2 ns = 11 ns

# CH 5 I/O

**Wired Connections**

- Limited factor for parallel: skew of voltage signals propagating in parallel lines: need to wait for all signals propagating on parallel lines to reach destination

Parallel (One wire per bit)

- ATA (Advanced Technology Attachment): for connecting computer with storage devices
- PCI (Peripheral Component Interface): connection between microprocessor and attached devices
- SCSI (Small Computer System Interface): set of parallel interface standard for attaching printers, disk drives, scanners, etc.

Serial (One wire per direction)

- SPI (Serial Peripheral Interface): synchronous

- I2C (Inter-Integrated Circuit): synchronous, multi-master, multi-slave, packet-switched, single-ended, serial bus
- USB (Universal Serial Bus): for cables and connectors for protocols for connection, communication, and power supply between computers and peripherals
- SATA (Serial ATA): connects mass storage devices, SSDs, optical drives

**Sensor**

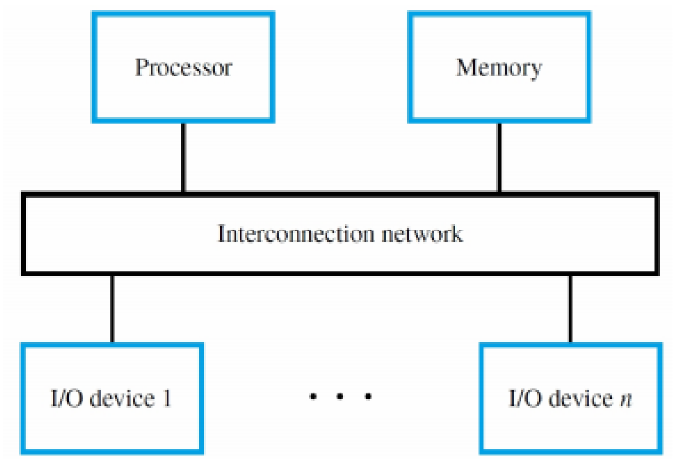
- Device measuring physical quantity

**Actuators**

- Device that alters a physical quantity

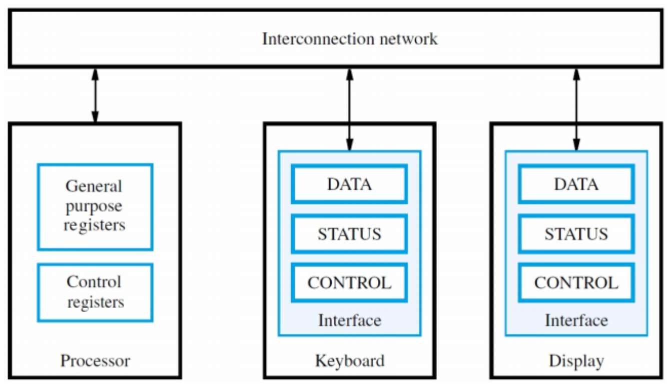
**Accessing I/O Devices**

- Memory-mapped I/O: accessed with Load and Store instructions
- Locations implemented as I/O registers within same address space



**I/O Device Interface**

- Circuit between device and interconnection network
- For data transfer, exchange of status and control information
- Includes data, status, and control registers accessible with load and store instructions
- Memory-mapped I/O: these registers viewed as memory locations



**I/O Synchronization**

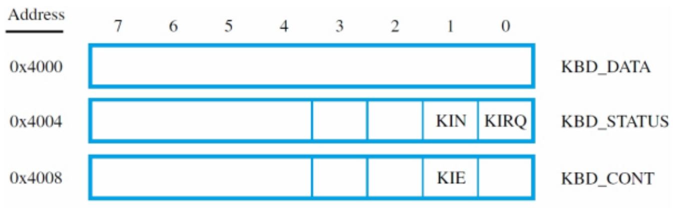
- Synchronization between devices and processor which operate at different speeds

**Signaling Protocol for I/O Devices**

- I/O devices can send "ready" signal (a status flag) to processor
- This status flag is polled by the processor

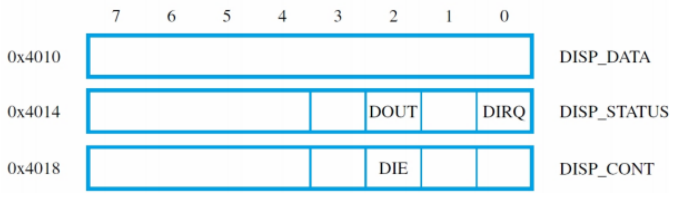
**Polled I/O Registers**

Example: Keyboard input



- Processor polls KBD_STATUS register to check KIN flag
- If KIN of KBD_STATUS is 1: processor reads KBD_DATA

Example: Display output



- Processor polls DISP_STATUS to check DOUT flag
- If DOUT is 1, processor writes to DISP_DATA

**Wait Loop**

- Loop for polling I/O device status
- Continues to poll until I/O device ready

Example:

```
READWAIT:    LDRB   R3, [R1, #4]  // read byte from KBD_STATUS
             TST    R3, #2        // check the value of KIN
             BEQ    READWAIT
             LDRB   R3, [R1]      // read from KBD_DATA
```

**Interrupts**

- Disadvantage of wait loop: processor is always occupied and cannot perform other useful computation
- Interrupt: let I/O device alert processor when it is ready
- Hardware sends interrupt-request signal to processor

**Interrupt-Service Routine**

- Executed at <u>any time</u> due to interrupt
- When processor sees interrupt: it completes current instruction, saves PC to temporary location, executes ISR (set PC to first instruction of ISR), and return from interrupt by restoring PC with previous address
- Save return address, processor and status registers since they can be changed by ISR. Usually the minimum (PC and status) registers are saved to reduce interrupt latency. ISR is responsible for saving any other registers
- Interrupt Latency: time before interrupt is serviced after it is raised

**Acknowledge of Interrupt**

- Signal from processor that tells device the interrupt has been recognized
- Device removes interrupt request
- Acknowledgement done by accessing status or data register in device interface

**Enable/Disable Interrupts**

- Global bit in processor that allows processor to acknowledge or ignore any interrupt from I/O devices
- A bit in an I/O device's interface's control register can disable interrupts from that device

**Event Sequence for an Interrupt**

- Processor status register has IE bit
- IE bit set to 1 to enable interrupts
- When interrupt recognized, processor saves PC and status register
- IE bit set to 0 to prevent additional interrupts
- After acknowledging and servicing interrupt, processor restore saved state and sets IE to 1 again

**Vectored Interrupts**

- Reduce service latency, no need to poll all I/O devices to see which one asserted an interrupt request
- Requesting I/O device identify itself by asserting a device-specific signal or sending a binary ID code to processor
- Processor finds correct ISR in an interrupt-vector table, which stores the address of the corresponding ISR (or a branch instruction to the ISR)
- Table lookup performed by hardware

**Priority**

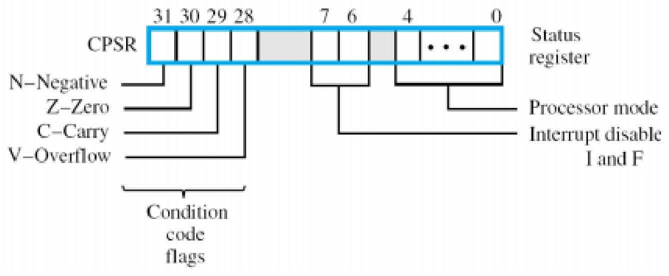
- Interrupts can have different priority
- Use hardware arbitration circuits to enforce desired priority across different I/O devices
- Must select only one device to provide index to the vector table

**Exceptions**

- Any interruption of execution

**ARM Processor Modes**

- 7 operating modes that determine what system resources a program has access to
- At the event of an interrupt:
  - IRQ mode: normal interrupt request
  - FIQ mode: fast interrupt request



**Banked Registers**

- Some registers in the range R8 – R14 are duplicated for different processor modes
- Allow more efficient switching by avoiding save/restore register operations
- IRQ: R13 and R14

- FIQ: R8 – R14

### ARM Return-from-Interrupt

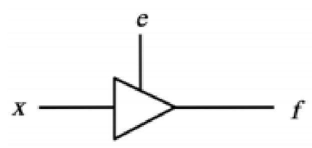
- Since processor pre-fetches next instruction, PC is incremented by 8
- To return from interrupt, processor must decrement address stored in LR by 4

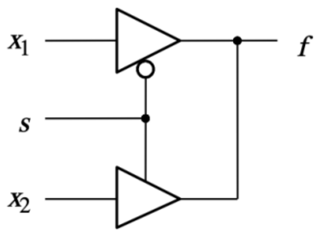
```
i      (currently executing instruction)
i+1
i+2    ← PC
```

### A Single-Bus System

- Bus: set of shared wires
- Only 1 pair of source/destination units can use bus to transfer data at a time
- Access managed by hardware to enforce this constraint

### Tri-State Buffers

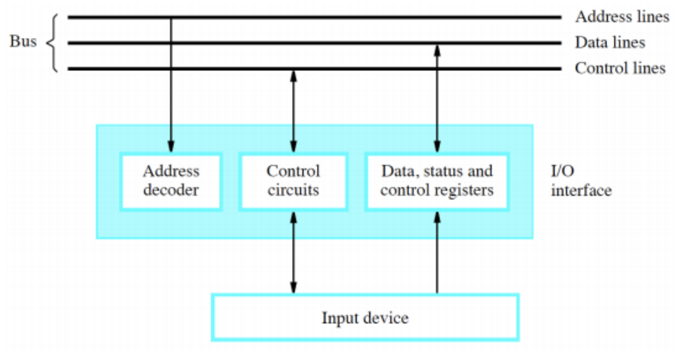




- Enable-controlled gate

### I/O Interface for Input Device

- Each I/O device assigned a unique set of addresses for its registers in its interface



### Bus Protocol

- Control signals indication action and when they are performed

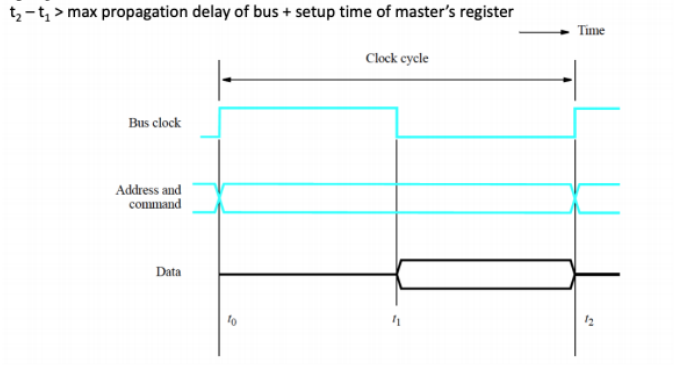
Bus Transfers

- Synchronous: all devices derive timing information from a <u>bus clock</u>
- Asynchronous

### Input Transfer Timing on Synchronous Bus

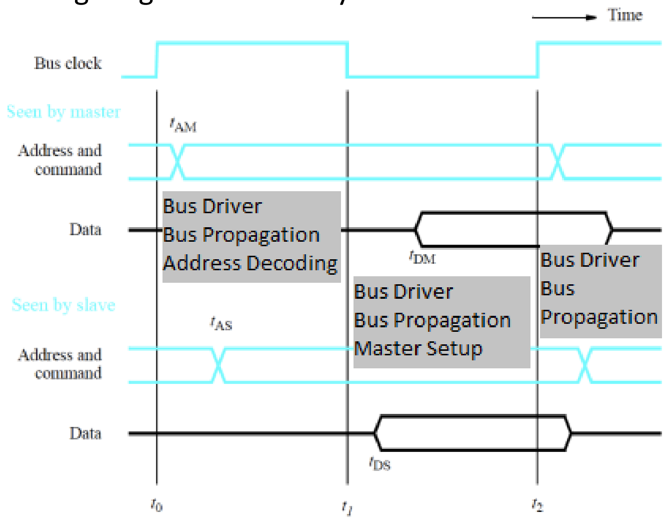
- Bus clock needs to be seen at all devices at the same time

$t_1 - t_0 >$ max. propagation delay of bus + time for slave to decode address and control signals
$t_2 - t_1 >$ max propagation delay of bus + setup time of master's register



- $t_2 - t_0$: must accommodate longest delays on the bus and slowest device interface
- All devices operate at speed of slowest device
- Delay introduced by <u>bus driver</u> which places new information on the address/data lines
- Delay encountered by <u>propagation</u> of information along the bus
- Delay introduced by <u>receiver</u> as it needs a minimum amount of <u>setup</u> time to receive data correctly

Timing Diagram with Delay

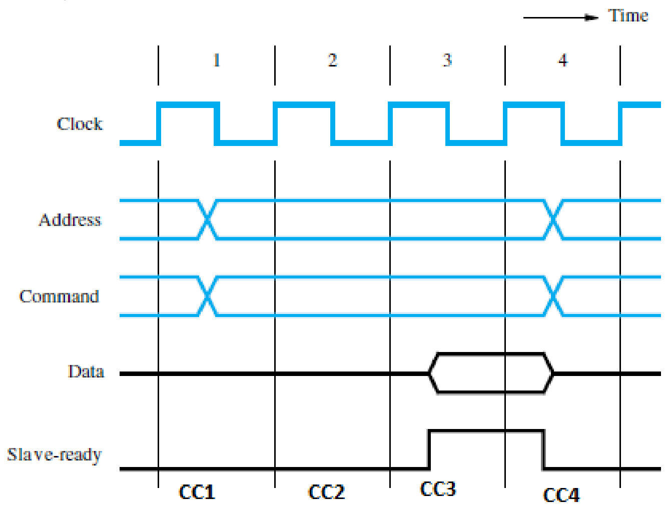


- $t_0$: master places device address on address lines and sends command on control lines indicating READ operation
- $t_1 - t_0$: information travels over bus, the clock pulse must be longer than the max propagation delay over the bus and the time for all devices to decode the address

- $t_1$: the addressed device (slave) responds by placing the requested input data on data lines
- $t_2 - t_1$: greater than the max propagation time on bus + setup time of the master's register
- $t_2$: master loads the data on data lines into one of its registers

**Multi-Cycle Data Transfer**

- Device response signal: indicates address was successfully decoded by device, and device is ready for data transfer
- Can be used to adjust the delay of a transfer operation: allow data transfer to span multiple cycles

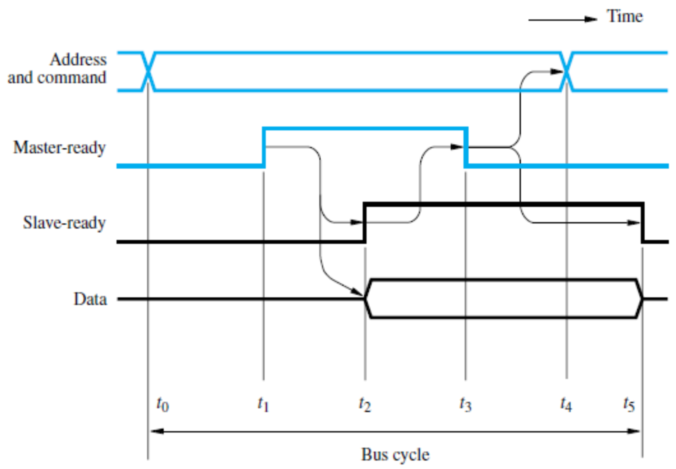


- CC1: Master sends address and command information on bus, slave receives information and decodes address
- CC2: Slave access requested data (assume delay)
- CC3: Slave places data on bus and asserts a slave-ready control signal. Master sees slave-ready signal and data on bus, loads data into registers
- CC4: Slave removes its data signals from bus and turns the slave-ready control signal low

**Asynchronous Protocol**

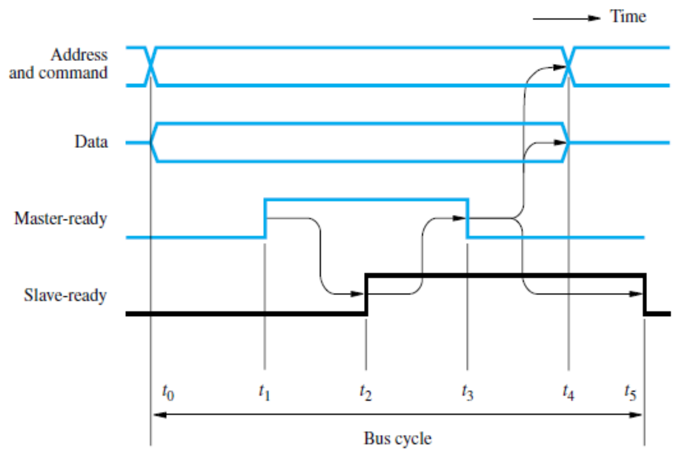
- No bus clock
- Handshake protocol: exchange of command and response signals between master and slave, <u>each signal change results in a response</u>. This is <u>full-handshake</u> or <u>fully-interlocked</u>

Handshake Control: Input Operation



- $t_0$: Master places address and command information on bus, all devices on bus decode this information
- $t_1 - t_0$: Bus driver delay + max bus propagation delay + address decoding by device
- $t_1$: Master sets master-ready to 1 to inform devices that command information is ready
- $t_2 - t_1$: Slave interface circuitry delay + max bus propagation delay
- $t_2$: Slave sees master-ready signal, performs required input operation by placing data on data lines, and sets the slave-ready signal to 1
- $t_3 - t_2$: max bus propagation delay + min master register setup time
- $t_3$: Slave-ready signal arrives at master, indicates that input data are available on bus. Master loads data into its register and drops master-ready signal, indicating data received
- $t_4 - t_3$: max bus propagation delay
- $t_4$: Master removes address and command information from bus
- $t_5$: Device interface receives 1-to-0 transition of master-ready signal, it removes data and slave-ready signal from bus

Handshake Control: Output Operation

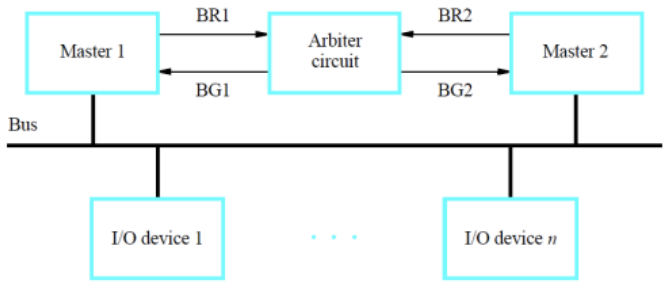


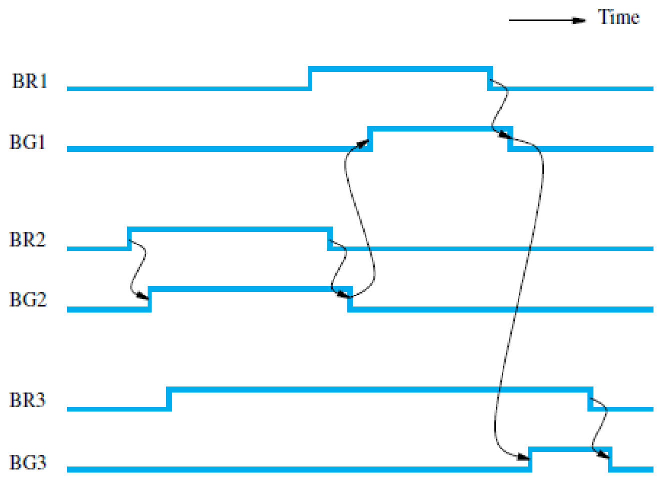
### Synchronous vs Asynchronous

| Synchronous | Asynchronous |
|---|---|
| Careful timing design | Automatic timing adjustment |
| 1 round trip delay | 2 round trip delay |
| Used in modern high-speed busses | |

### Arbitration

- Controlling access to shared resources
- Bus Master: Initiates read/write requests
- Arbiter: Grants bus to highest priority device
- Control lines on bus used to request and grant bus access





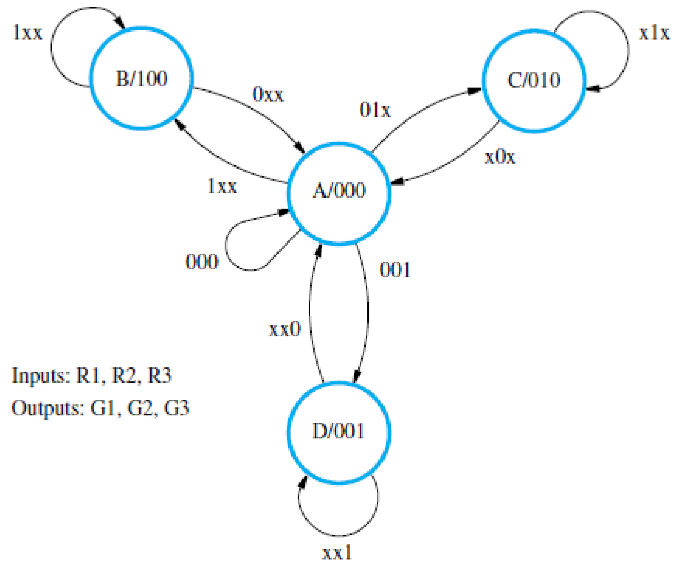
Priority: BR1 > BR2 > BR3

- At first, master 2 sends BR. Since there are no other BR, arbiter grants master 2 bus access
- When master 2 finished using bus, it lowers the BR signal. During the time of use, arbiter received BR from master 3 first then master 1. Since master 1 has higher priority, arbiter grants bus access to master 1.
- When master 1 finishes using bus, it lowers its BR signal. Now the only BR is from master 3, the arbiter grants bus access to master 3.

Example:

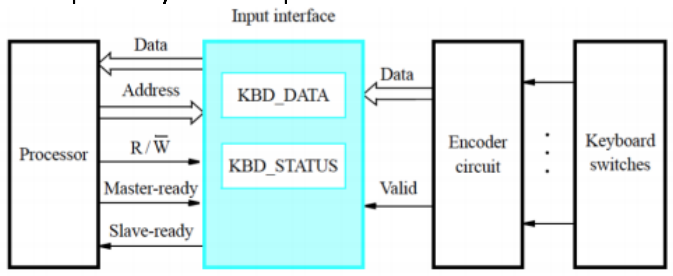
**Problem:** An arbiter receives three request signals, R1, R2, R3, and generates three grant signals, G1, G2, G3. Request R1 has the highest priority and request R3 the lowest priority. An example of the operation of such an arbiter is given in Figure 7.9. Give a state diagram that describes the behavior of this arbiter.

**Solution:** A state diagram is given in Figure 7.21. The arbiter starts in the idle state, A. When one or more of the request signals is asserted, the arbiter moves to one of the three states, B, C, or D, depending on which of the active requests has the highest priority. When it enters the new state, it asserts the corresponding grant signal. The arbiter remains in that state



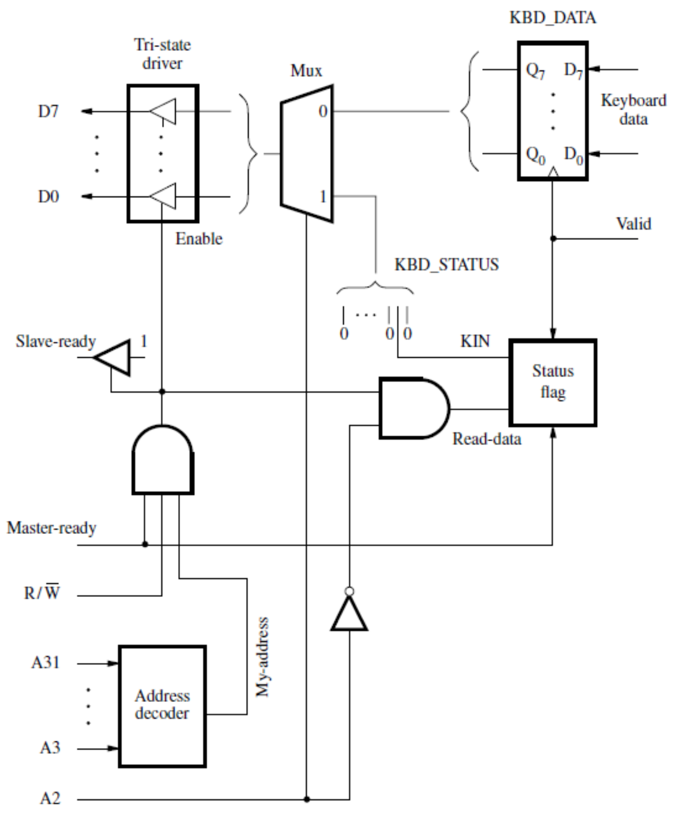
### I/O Interface Circuits

Example: Keyboard Input to Processor

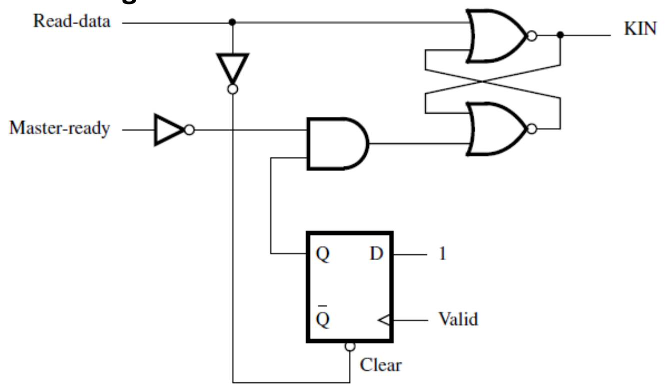


- Valid bit changes from 0 to 1: KBD_STATUS.KIN = 1 and KBD_DATA is loaded with data
- Processor reads KBD_DATA and KBD_STATUS.KIN is set to 0

### Data & Status Info

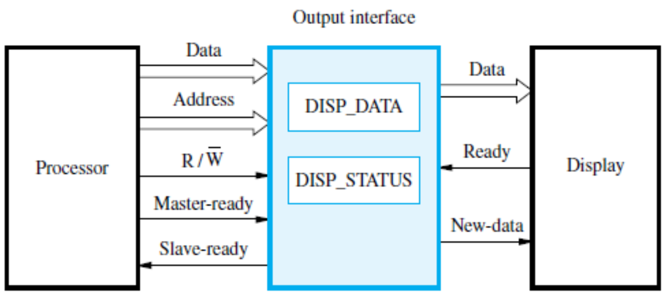


## Status Flag Control

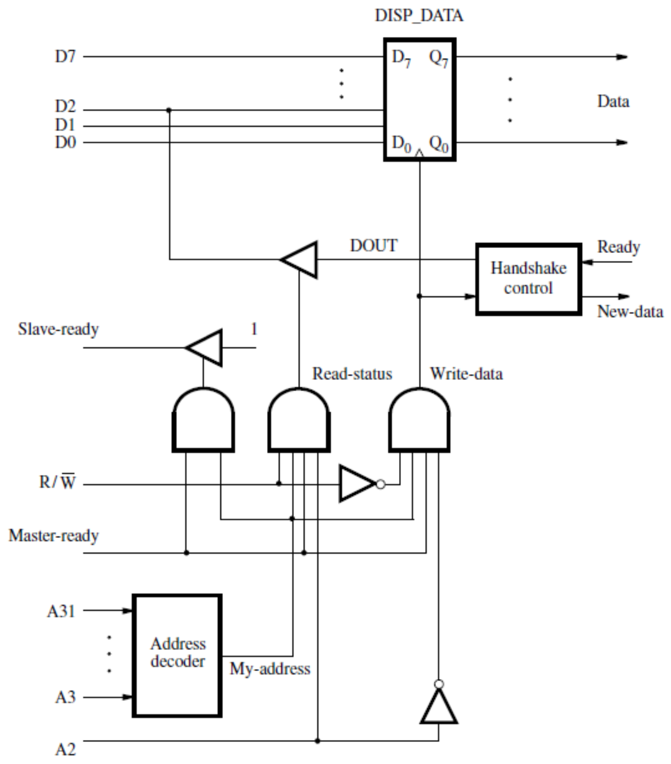


- When valid is 1, KIN is 1 only if master-ready is low. Purpose is to ensure that KIN does not change state when being read by the processor
- Both NOR latch and flip-flop are set to 0 when read-data is 1, indicating that KBD_DATA is being read

## Output Interface

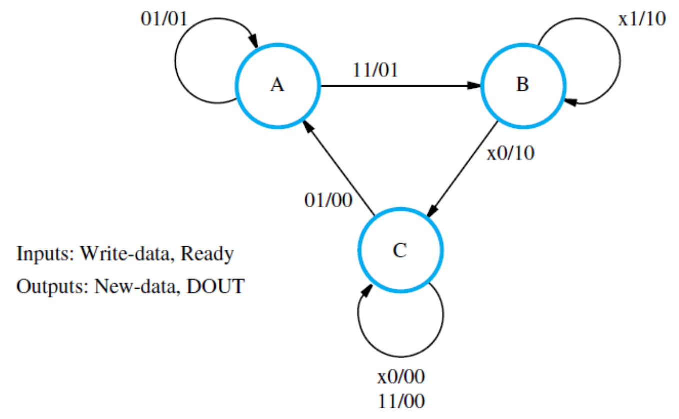


- Display: asserts ready signal, DISP_STATUS.DOUT is 1
- Processor checks and see DISP_STATUS.DOUT is 1, sends data to DISP_DATA, this sets DOUT to 0 and New-data to 1
- Display sets Ready to 0, accepts and displays the data in DISP_DATA



- Ready and New-data signals form handshakes

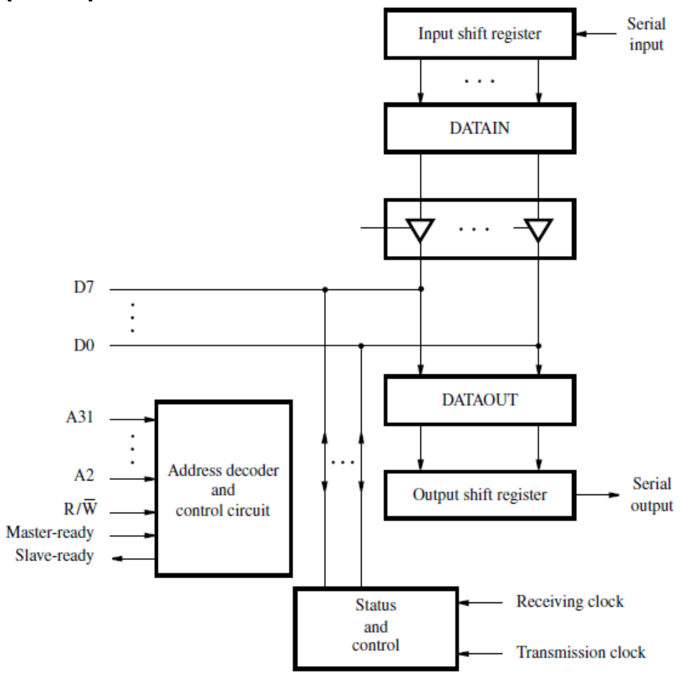
## Handshake Control Circuit FSM



### Serial Links

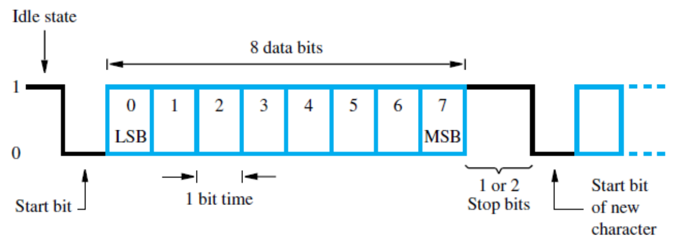
- Serial data transmission: more suitable for longer distances, less expensive
- Data transmitted one bit at a time

### Universal Asynchronous Receiver Transmitter (UART)



- Input shift register accepts 8 serial bits and converts them to parallel, when all bits are received, data is loaded into DATAIN register
- Implementation of separate DATAIN and shift registers (double buffering) allows continuous transfer of characters, the transfer of the second character can begin before the processor reads the first one

### Start-Stop Transmission (Asynchronous)



- Idle state: 1 bit high
- Start bit: 1 bit low
- Stop bit: 1 or 2 bits high
- Falling edge from idle state to start bit alerts receiver
- Transmitter and receiver use separate clocks, typically $f_R = 16 f_T$, with substantially higher clock rate, in order to sample as close to the center of each bit as possible
- Module-16 counter used.
  - Counter resets when leading edge of start bit detected.
  - At the count of 8, the middle of the start bit is reached. The input line state is sample again to verify and the counter is cleared to 0 again.
  - From this point onward, the input is sampled whenever count 16 is reached.
  - Thus, as long as $\frac{f_R}{16}$ is close to $f_T$, the data loaded will be correct.

### Synchronous Transmission

- For high-speed
- Receiver generates a clock synchronized to the transmitter by observing bit pattern at the beginning of the transmission
- Active edge of clock adjusted to be in the center of the bit position

### USB Speeds

- Low-Speed: 10 – 100 kb/s
  - 1.5 Mb/s signaling bit rate
- Full-Speed: 500 kb/s – 10 Mb/s
  - 12 Mb/s signaling bit rate
- High-Speed: 400 Mb/s
  - 480 Mb/s signaling bit rate
- SuperSpeed: 4Gb/s
  - 5Gb/s signaling

**USB Features**

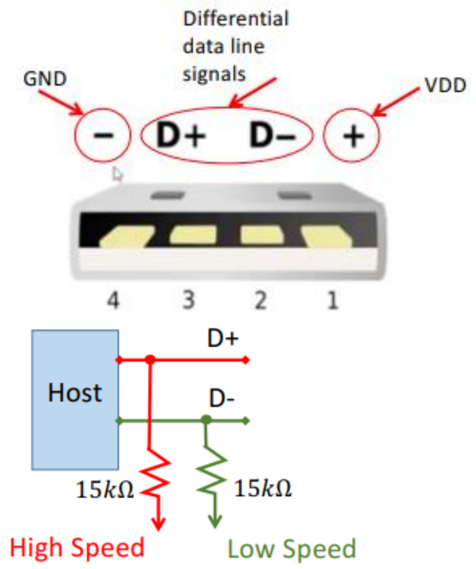
- Able to send isochronous (periodic) and asynchronous data over a common link
- Point-to-Point connection using serial transmission and two twisted pairs (+5V, ground, two data wires)
- Low-speed transmission: single-ended, one data wire for 0 and one for 1
- High-speed transmission differential signaling
  - Data encoded as voltage difference between two data wires
  - Noise cancelled as it is common to both wires

**USB Connectors**

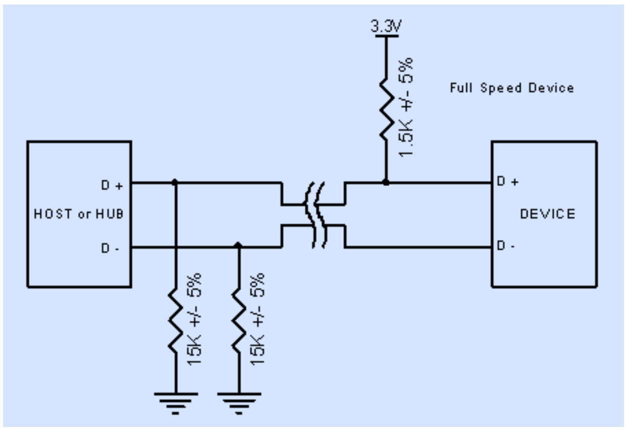
- Series A cable: connects USB devices to hub port
- Series B cable: connects detachable devices (hot-swappable)
- Mini-USB
- Micro-USB

**USB Protocol**

- USB pulls one of data lines (D+ or D-) high using 15 kOhm resistors
- Low-speed devices must pull D- line high to identify its speed
- High-speed devices must pull D+ line high to identify its speed

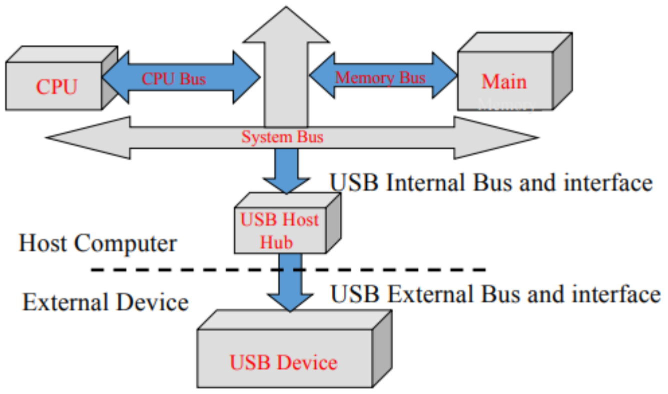


Example: High speed



- With the D+ pulled high, the host/hub will see a high D+ voltage when USB plugged-in (resistor divider)

**Bus Hierarchy**



**Bus Topology**

- Connects computer to peripheral devices
- All devices linked to common point (root hub)
- USB polls devices to resolve simultaneous messages. All transfers are initiated by host through polling to avoid interference.
- USB devices assigned with 7-bit address: 127 different devices
- USB devices have their own address space, local to the USB tree, and do not share the same address space with the processor

**USB Host**

- Contains a sophisticated set of software drivers
- Drivers schedule and compose USB transactions and access individual devices to obtain configuration information
- Difficult to use on standalone systems without OS support

- Physical interface (chip) to USB root hub is called the USB Host Controller

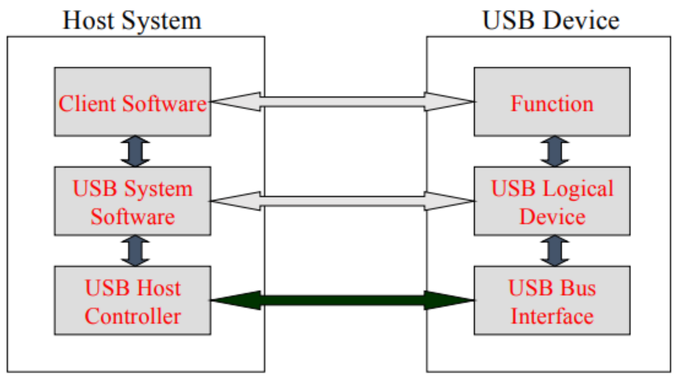
### USB Hub

- Tests new devices and maintains status information of child devices
- Serve as repeaters, boosting strength of up and downstream signals
- Electrically isolates devices: allow expanded number of devices; slower devices to be placed on faster branch; malfunctioning devices to be removed

### USB Devices

- All functioning devices are slaves
- Many indicate need to transmit/receive data through polling
- Contain registers that identify relevant configuration information

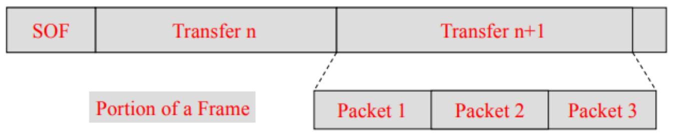
### USB Software Interfaces



- Client software: determine what kind of data to be transferred
- USB system software: scheduling and configuration of data transfers
- USB host controller: composes and regulates data transfer, keeps track of sent and received data

### Frames, Transfers, and Packets

- Bandwidth: composed by host into 1ms time periods (one frame)
- Each frame: composed of a sequence of transactions to occur within given time period
- Each transaction: composed of a sequence of packets that outline the format and corresponding data for each transaction
- A turn-around time may exist between transfers of each packet
- Data configured by host to send out within a given frame
- Start of frame indicated by SOF packets
- Host controller can modulate frame length for more efficient use of bandwidth
- Transactions initiated with a token packet, followed by a data packed, and concluded with a handshaking packet
- Lengthy transactions broken into smaller ones and sent over a number of frames

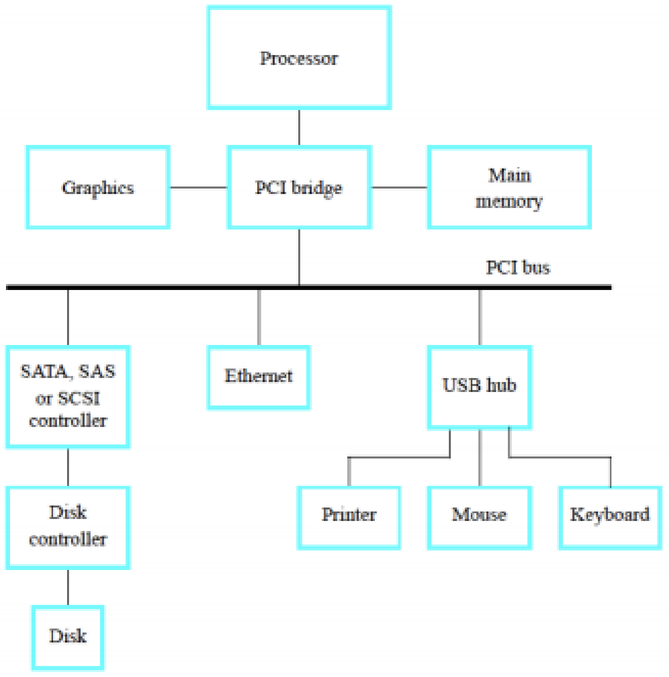


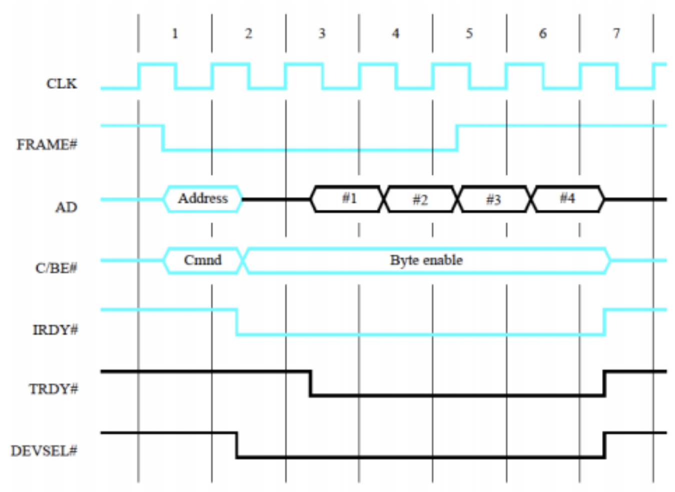
### USB Packet Encoding

- Data Fields: 0 – 1023 bytes (10 bits), must have an integral number of bytes. Data within each byte are shifted out LSB first
- End of Packet Fields: Both differential lines driven to 0 for two clock cycles and then one of them to 1 for one clock cycle

### PCI (Bus)

- Processor independent
- Devices on PCI bus appear in the address space of the processor



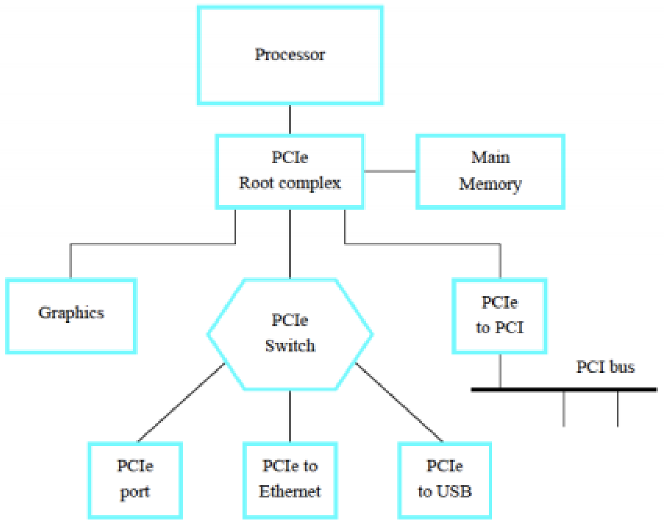


### Plug-and-Play

- Possible by bus's initial connection protocol
- Up to 21 device connectors on PCI bus
- Each PCI-compatible device has small ROM with information on device characteristics
- Processor scans all connectors to determine whether a device is plugged in
- Processor assigns address to each device and reads its ROM, then it selects appropriate device driver software and performs initialization

### PCIexpress

- Point-to-point connections with one or more switches forming a tree
- Root complex provide high-speed ports for memory and other devices



### PCIe Links

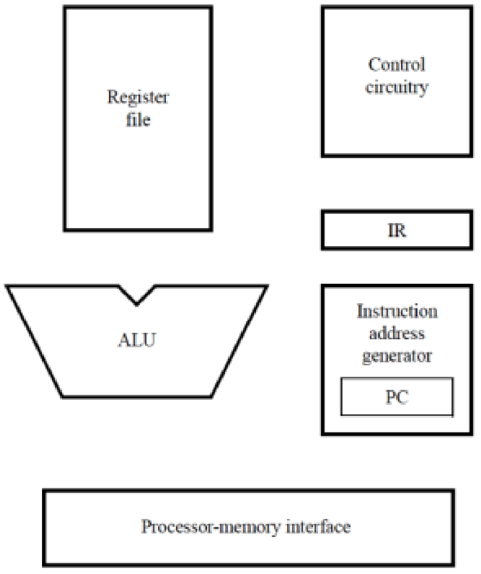
- Basic connection: lane
- Each lane: two twisted-pairs or optical lines for each direction of transmission
- Data rate: 2.5 Gb/s in each direction
- A connection to a device (link) may use up to 16 lanes
- The PCIe protocols are fully compatible with PCI as they use the same initial connection protocol

# CH 6 Processor

### Processor

- Fetches one instruction at a time
- Decodes the instruction
- Carries out actions specified by the instruction

### Processor Building Blocks



- PC: address of next instruction
- IR: instruction fetched into IR
- Instruction address generator: updates PC
- Control Circuitry: decodes instructions and generate control signals that direct datapath

### Instruction Execution

- In RISC machine: all instructions executed in the same number of steps, each step in separate hardware stage
- RISC processor design: 5 hardware stages

General Process:

1. Fetch & update PC
2. Decode & register R/W
3. ALU computation

4. Memory access
5. Register R/W

| Step | Action |
|---|---|
| 1 | Fetch an instruction and increment the program counter. |
| 2 | Decode the instruction and read registers from the register file. |
| 3 | Perform an ALU operation. |
| 4 | Read or write memory data if the instruction involves a memory operand. |
| 5 | Write the result into the destination register, if needed. |

Example: LDR R5, [R7, R8]

1. Fetch the instruction and increment the program counter
2. Decode the instruction and read the contents of registers R7 and R8 in the register file
3. Compute the effective address
4. Read the memory source operand
5. Write the operand into the destination register

Example: ADD R3, R4, R5

1. Fetch the instruction and increment the program counter
2. Decode the instruction and read registers R4 and R5 from the register file
3. Compute the sum
4. No action
5. Write the result into the destination register

Example: ADD R3, R4, #1000

- Immediate operand given in instruction word and stored in IR

1. Fetch the instruction and increment the program counter
2. Decode the instruction and read register R4 from the register file
3. Compute the sum
4. No action
5. Write the result into the destination register

Example: LDR R5, [R7, #X]

- Immediate operand given in instruction word and stored in IR

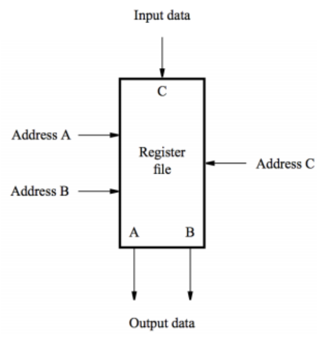
1. Fetch the instruction and increment the program counter
2. Decode the instruction and read the contents of register R7 in the register file
3. Compute the effective address
4. Read the memory source operand
5. Write the operand into the destination register

Example: STR R6, [R8, #X]

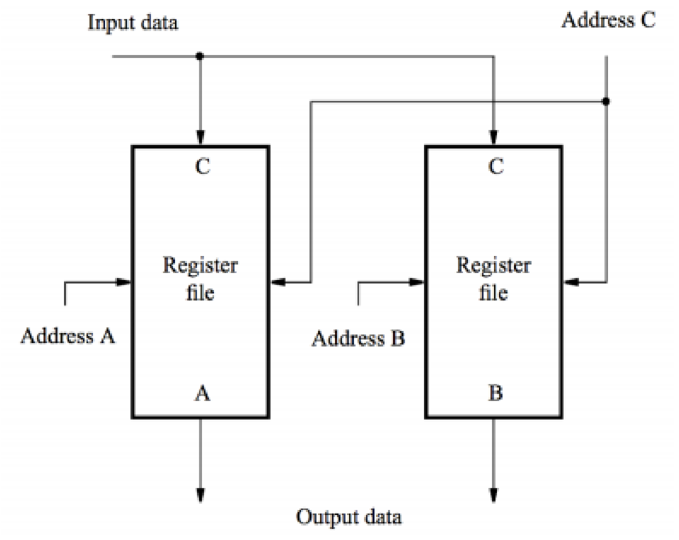
1. Fetch the instruction and increment the program counter
2. Decode the instruction and read the contents of registers R6 and R8 in the register file
3. Compute the effective address
4. Store the contents of register R6 into memory location X + [R8]
5. No action

**Register File**

- 2-port register file needed to read 2 source registers at the same time
- Implemented using 2-port memory



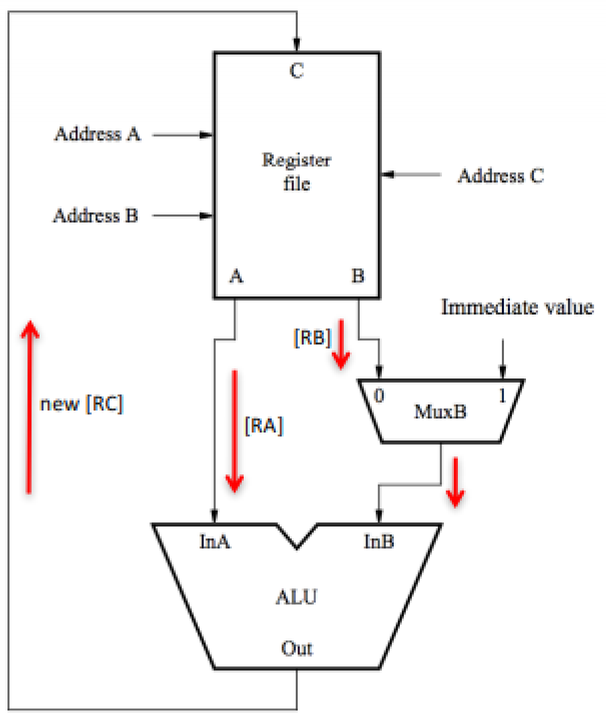
Or using 2 single-ported memory



**Conceptual View**

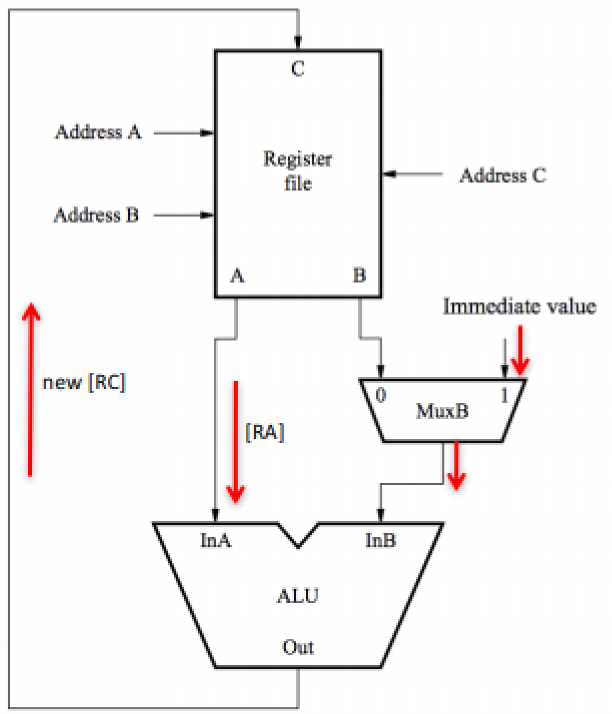
Computational Instructions

- Both source operands and destination location are in register file
- [RA] and [RB]: values of registers at addresses A and B
- [RC]: result stored to register at address C



Immediate Instructions

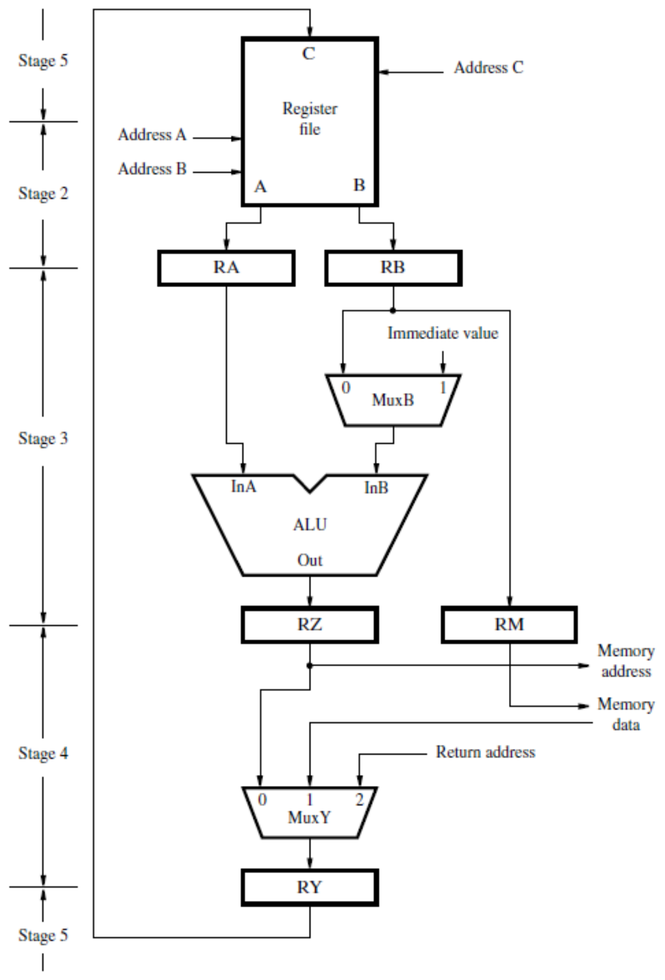
- One of the source operand is in IR



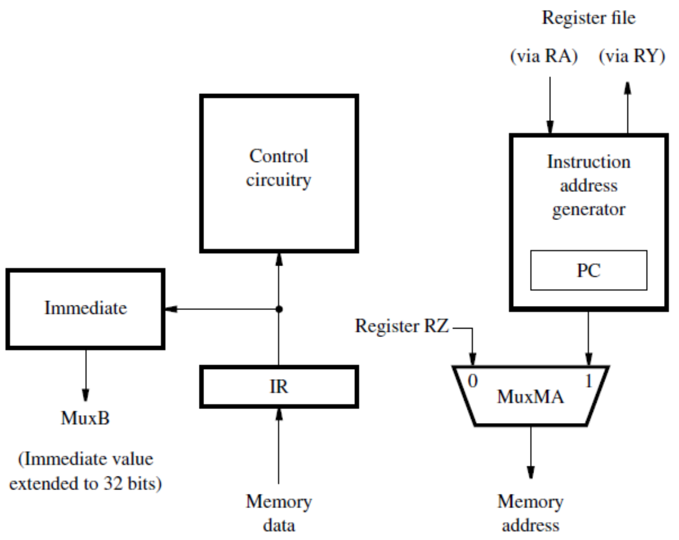
**Datapath**

- Operations within each stage are completed within 1 clock cycle
- Inter-stage registers: registers to transfer data across stages

- Register file spans across stages 2 and 5 since it contains both source registers and destination register

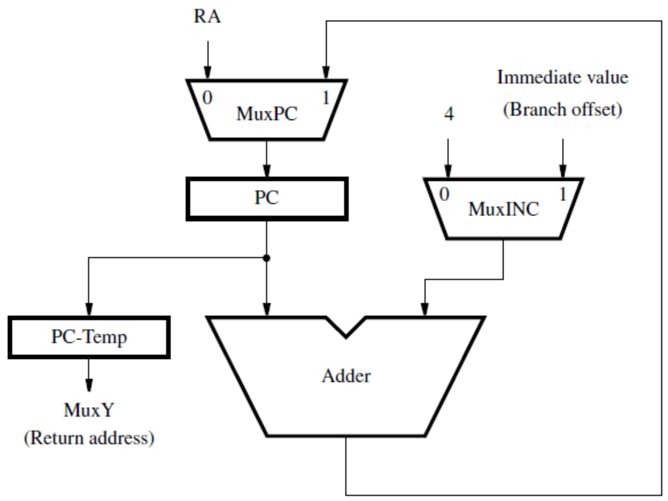


**Instruction Fetch Section**



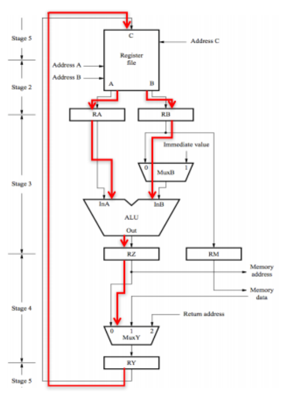
- When an instruction is read, it is placed into IR
- Control circuitry decode instruction and generates control signals that drive all units

Instruction Address Generator Circuit



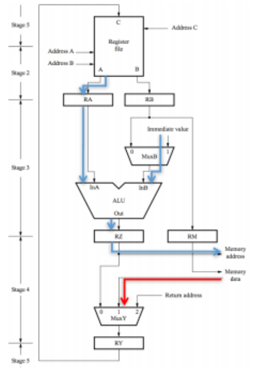
Example: Add R3, R4, R5

1. Memory address ← [PC],
   Read memory,
   IR ←Memory data,
   PC ← [PC] + 4
2. Decode instruction,
   RA ← [R4],
   RB ← [R5]
3. RZ ← [RA] + [RB]
4. RY ← [RZ]
5. R3 ← [RY]



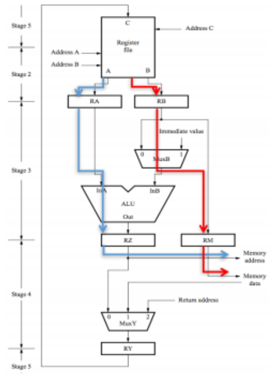
Example: Load R5, [R7, #X]

1. Memory address ← [PC],
   Read memory,
   IR ← Memory data,
   PC ←[PC] + 4
2. Decode instruction,
   RA ←[R7]
3. RZ ←[RA] + Immediate value X
4. Memory address ←[RZ],
   Read memory,
   RY ← Memory data
5. R5 ←[RY]



Example: STR R6, [R8, #X]

1. Memory address ←[PC],
   Read memory,
   IR ← Memory data,
   PC ← [PC] + 4
2. Decode instruction,
   RA ←[R8],
   RB ←[R6]
3. RZ ←[RA] + Immediate value X,
   RM ←[RB]
4. Memory address ←[RZ],
   Memory data ←[RM],
   Write memory
5. No action



**Unconditional Branch**

1. Memory address ←[PC],
   Read memory,
   IR ← Memory data,
   PC ←[PC] + 4
1. Decode instruction
2. PC ←[PC] + Branch offset
3. No action
4. No action

**Conditional Branch**

- For generic RISC branch that do not use conditional codes: processor compare values in registers and tests for some conditions

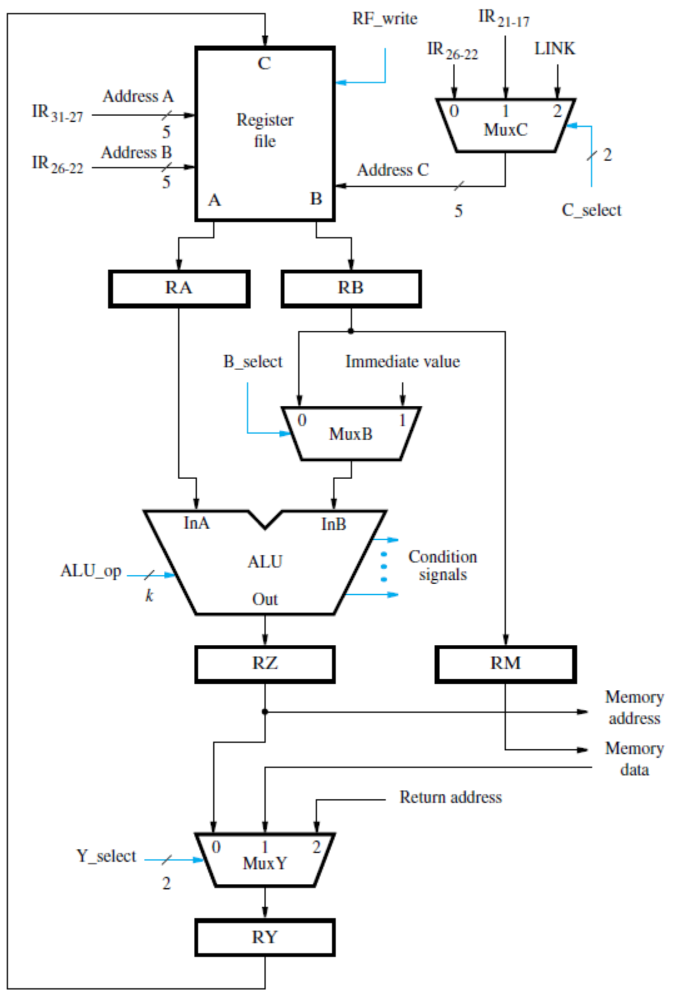
Conditional branch: BEQ R5, R6, LOOP

1. Memory address ←[PC],
   Read memory,
   IR ← Memory data,
   PC ←[PC] + 4
2. Decode instruction,
   RA ←[R5],
   RB ←[R6]
3. Compare [RA] to [RB],
   If [RA] = [RB], then PC ←[PC] + Branch offset
4. No action
5. No action

- The ALU performs subtraction and control circuits test result both in one clock cycle

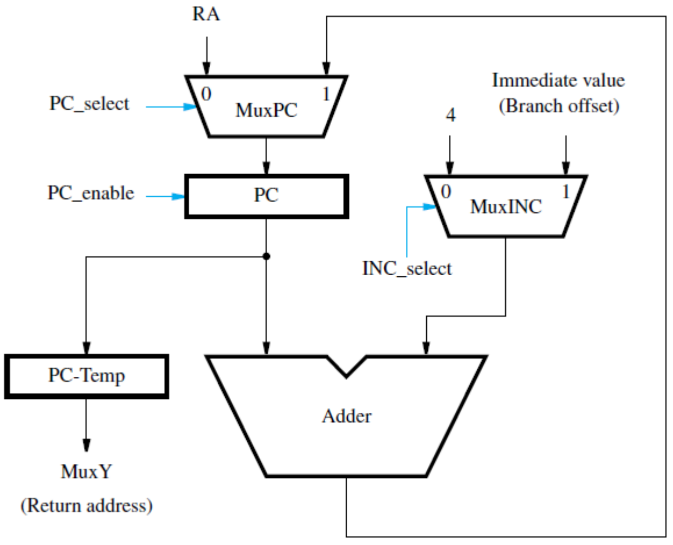
**Control Signals**

- Inter-stage registers are always enabled since data are transferred from one stage to the next in every clock cycle
- RF_write: new data are loaded into the selected register only when the control signal RF_write is asserted
- C_select: selects destination register
- B_select: selects second source operand either register or immediate value
- ALU_op: selects ALU operation
- Y_select: selects the result to be stored in the destination register



1.Memory address ← [PC], Read memory, Wait for MFC, IR ← Memory data, PC ←[PC] + 4

**Control Signals of Instruction Address Generator**



Control Signal Generation

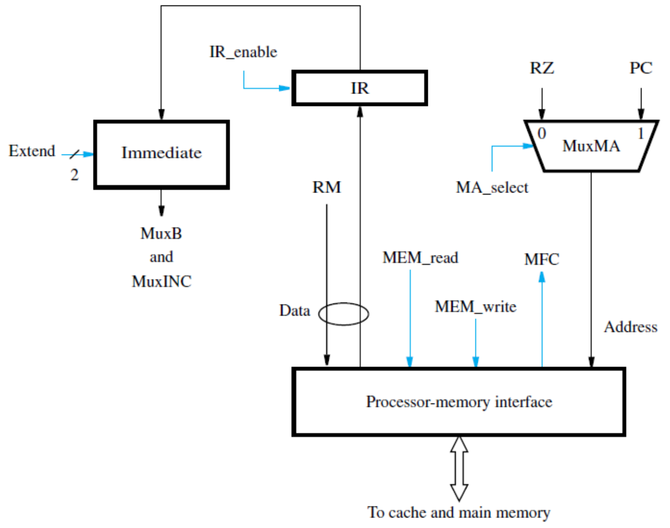
- Control signals generated to ensure actions in datapath take place in correct sequence at correct time
- Two approaches: hardwire control and microprogramming

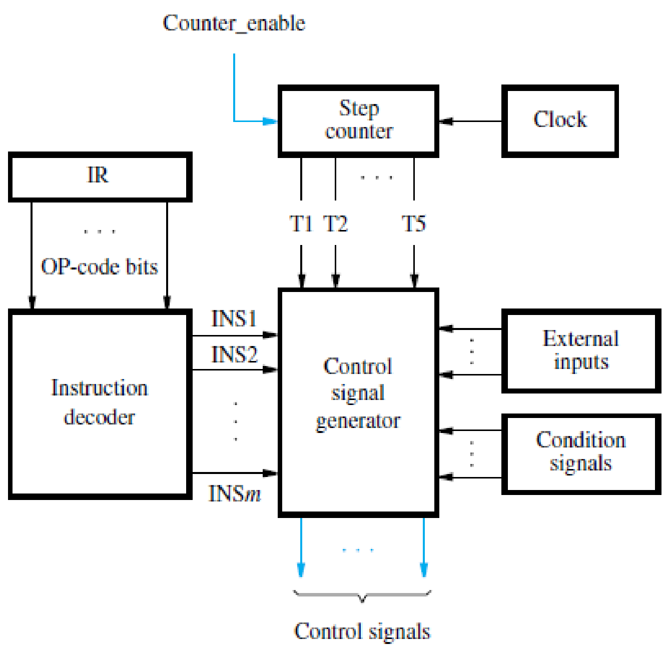
Hardware Control (FSM):

- Implementing circuitry that considers step counter, IR, ALU result, and external inputs
- States of FSM kept in counter that keeps track of execution step
- Outputs are control signals

**Memory Access**

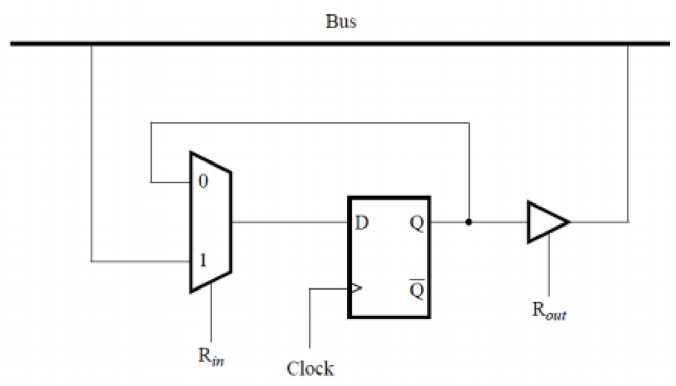
- When data are found in cache, access to memory can be completed in 1 clock cycle
- Read and write from main memory may require several clock cycles
- Control signal to indicate memory function completion is needed (MFC)



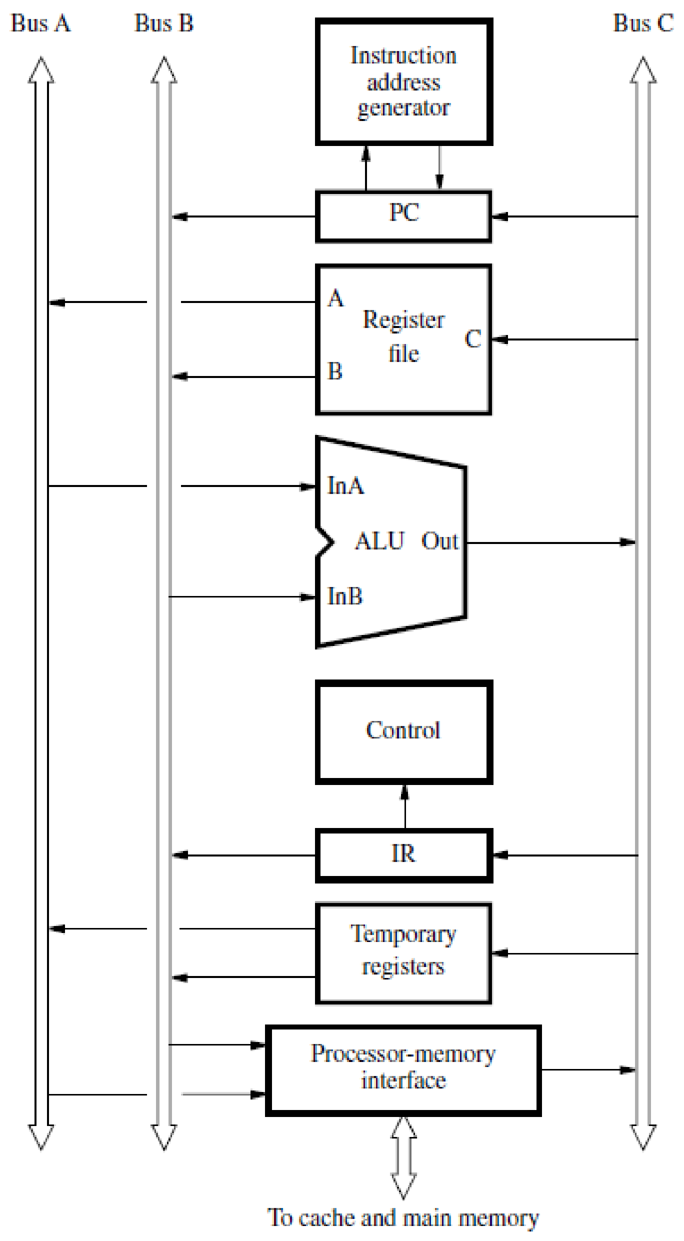


## Bus

- Tri-state drivers needed when functional units connected to a common bus



## 3-Bus Interconnection Network



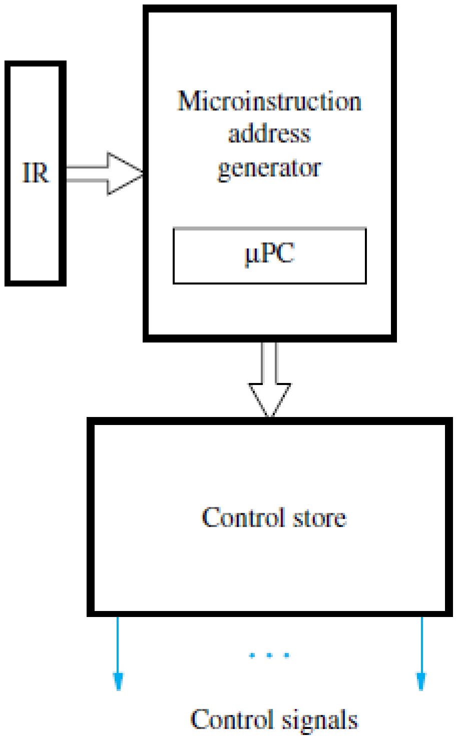
Example: Add R5, R6

1. Memory address ←[PC],
   Read memory,
   Wait for MFC (Memory Function Completed signal),
   IR ← Memory data,
   PC ←[PC] + 4
2. Decode instruction
3. R5 ←[R5] + [R6]

## Microprogramming

- Software-based approach for the generation of control signals
- Values of control signals for each period stored in a microinstruction (control word)

- A processor instruction is implemented by a sequence of microinstruction that are placed in a control store
- From decoding of an instruction in IR, the control circuitry executes the corresponding sequence of microinstructions
- MicroPC maintains the location of current microinstruction
- Provides flexibility needed to implement more complex instructions in CISC processors; however, reading and executing microinstructions incur undesirably long delays in high-performance processors


IR
Microinstruction address generator
μPC
Control store
. . .
Control signals